Lab 02: Making Your Project Functional with Data and Experiment Tracking

Overview

Welcome to Lab 02! In this lab, you'll transform your well-structured ML project from Lab 01 into a fully functional, production-ready system with proper environment management, command-line interfaces, data versioning, and experiment tracking.

In Lab 01, you created a clean project structure and modularized your code. Now it's time to add the tools and practices that make your project:

- **Reproducible** - Anyone can recreate your environment and results
- **Accessible** - Command-line interfaces make your code easy to use
- **Trackable** - Version control for data and experiments
- **Testable** - Automated tests ensure code quality
- **Production-ready** - Ready for deployment and collaboration

Learning Objectives

By the end of this lab, you will:

1. **Environment Management**: Understand computational environments and create isolated Python virtual environments
2. **Dependency Management**: Create and manage `requirements.txt` files
3. **CLI Development**: Build command-line interfaces using argparse
4. **Best Practices**: Implement hyperparameter tuning and scikit-learn pipelines
5. **Configuration Management**: Use YAML files for flexible configuration
6. **Data Versioning**: Use DVC (Data Version Control) with DagsHub
7. **Experiment Tracking**: Use MLflow to track experiments and model performance
8. **Testing**: Write unit tests with pytest

Prerequisites

- Completed Lab 01 (modular project structure)
- Python 3.12.x installed
- Git repository set up
- Basic understanding of command line

Part 1: Understanding Computational Environments

What is a Computational Environment?

A **computational environment** is the complete software ecosystem where your code runs, including:

- Operating system
- Python interpreter version

- Installed packages and their versions
- System libraries
- Environment variables

The "It Works on My Machine" Problem

Consider this scenario:

```
You: "My code works perfectly!"
Teammate: "It crashes on my machine..."
You: "But it works for me! 🤔"
```

Why does this happen?

- Different Python versions (3.10 vs 3.12)
- Different package versions (numpy 1.24 vs 2.3)
- Missing dependencies
- Different operating systems

Why Standardized Environments Matter

In Development:

- Ensures code works identically for all team members
- Prevents "dependency hell"
- Makes onboarding new developers easier
- Enables reproducible research

In Production:

- Guarantees consistent behavior in deployment
- Enables rolling back to previous versions
- Facilitates automated testing and CI/CD
- Prevents production failures due to environment mismatches

The Solution: Containerization and Virtual Environments

Long-term solution (coming in Lab 04): **Docker**

- Containers package your code AND its entire environment
- Works identically everywhere (local, cloud, different OS)
- Industry standard for deployment

Today's solution: Python **Virtual Environments**

- Isolated Python environment for your project
- Project-specific package versions
- Prevents conflicts between projects
- Lightweight and easy to use

Why start with virtual environments? Our project currently uses only Python and doesn't require system-level dependencies. A Python virtual environment is sufficient for now. Later, when we deploy to production or need to ensure OS-level consistency, we'll containerize with Docker.

Part 2: Creating and Managing Virtual Environments

What is a Virtual Environment?

A Python virtual environment is an isolated Python installation that:

- Has its own Python interpreter copy
- Has its own `site-packages` directory (where packages install)
- Doesn't interfere with system Python or other projects
- Can have different package versions per project

Analogy: Think of it as a separate apartment for each project - each has its own furniture (packages) and doesn't share with others.

Creating a Virtual Environment

Python includes the `venv` module for creating virtual environments:

```
# Create a virtual environment named .venv
python -m venv .venv
```

Breaking down the command:

- `python` - Run Python interpreter
- `-m venv` - Run the venv module as a script
- `.venv` - Name of the directory to create

Why `.venv` (with a dot)?

- Hidden directory (dot prefix on Unix/Linux/Mac)
- Industry convention
- Most IDEs auto-detect `.venv`
- Clearly indicates it's a virtual environment

What gets created?

```
.venv/
├── bin/              # Executables (Mac/Linux)
│   ├── python        # Python interpreter copy
│   ├── pip           # Package installer
│   └── activate      # Activation script
├── Scripts/          # Executables (Windows)
│   ├── python.exe
│   ├── pip.exe
```

```
│   └── activate.bat
├── lib/              # Installed packages
│   └── python3.12/
│       └── site-packages/
└── pyvenv.cfg        # Configuration
```

Activating the Virtual Environment

On Mac/Linux:

```
source .venv/bin/activate
```

On Windows (Command Prompt):

```
.venv\Scripts\activate.bat
```

On Windows (PowerShell):

```
.venv\Scripts\Activate.ps1
```

How to tell it's activated? Your command prompt changes:

```
# Before activation
user@computer:~/project$

# After activation
(.venv) user@computer:~/project$
```

The `(.venv)` prefix indicates you're in the virtual environment!

Verify Activation

```
# Check which Python is being used (should point to .venv)
which python    # Mac/Linux
where python    # Windows

# Output should be something like:
# /path/to/your/project/.venv/bin/python

# Check Python version
python --version
# Python 3.12.12
```

```
# List installed packages (should be minimal at first)
pip list
# Package    Version
# ---------- -------
# pip        24.0
# setuptools 65.5.0
```

Deactivating the Virtual Environment

When you're done working:

```
deactivate
```

Your prompt returns to normal:

```
# After deactivation
user@computer:~/project$
```

Best Practices

✅ **DO**:

- Create one virtual environment per project
- Name it `.venv` (convention)
- Activate before installing packages
- Commit requirements.txt, NOT the .venv folder
- Document activation steps in README

❌ **DON'T**:

- Commit the `.venv` folder to Git
- Share virtual environments between projects
- Install packages globally
- Forget to activate before working

Adding .venv to .gitignore

Ensure your `.gitignore` includes:

```
# Virtual environments
.venv/
venv/
env/
ENV/
```

This prevents the (large!) virtual environment from being committed to Git.

Part 3: Dependency Management with requirements.txt

Why Document Dependencies?

Imagine you want to share your project:

```
# Your code
import pandas as pd
import sklearn
import some_package_you_installed_6_months_ago
```

Without documentation:

- Users don't know what packages to install
- Users don't know which versions you used
- Code may break with different versions

With requirements.txt:

- Clear list of all dependencies
- Exact or compatible versions specified
- One command installs everything
- Reproducible environment

The Two Approaches

Approach 1: pip freeze (Automatic)

```
# Activate your virtual environment
source .venv/bin/activate

# Generate requirements.txt with all installed packages
pip freeze > requirements.txt
```

Example output:

```
catboost==1.2.8
certifi==2025.1.12
charset-normalizer==3.4.1
contourpy==1.3.1
cycler==0.12.1
dvc==3.63.0
fonttools==4.55.3
idna==3.10
```

```
joblib==1.5.2
kiwisolver==1.4.7
...many more...
threadpoolctl==3.6.0
tzdata==2025.2
urllib3==2.3.0
```

Pros:

- Quick and automatic
- Captures exact versions
- Includes all dependencies

Cons:

- Lists sub-dependencies (packages you didn't directly install)
- Hard to read and understand
- Can cause conflicts
- Difficult to maintain

Approach 2: Hand-Curated (Recommended)

Manually create `requirements.txt` with only direct dependencies:

```
# Core ML and Data Science
numpy==2.3.4
pandas==2.3.3
scikit-learn==1.7.2

# Model Persistence
joblib==1.5.2

# Visualization
matplotlib==3.10.7
seaborn==0.13.2
plotly==6.3.1

# Advanced Models
xgboost==3.1.1
catboost==1.2.8

# Configuration
PyYAML==6.0.3

# Data Version Control (DagsHub remote - recommended)
dagshub==0.6.3
dvc-s3==3.2.2

# Testing
pytest==8.4.2
pytest-cov==7.0.0
```

```
# Development
jupyter==1.1.1
```

Pros:

- Clean and readable
- Only direct dependencies
- Easy to maintain
- Documents what you actually use
- Less likely to cause conflicts

Cons:

- Requires manual maintenance
- Need to remember to update when adding packages

Version Pinning Strategies

Exact pinning (==):

```
pandas==2.3.3  # Exact version
```

- Most reproducible
- Safest for production
- May miss bug fixes

Compatible release (~=):

```
pandas~=2.3.3  # Compatible: >=2.3.3, <2.4.0
```

- Allows patch updates
- Balance of stability and updates

Minimum version (>=):

```
pandas>=2.3.0  # Any version 2.3.0 or higher
```

- Most flexible
- Can break compatibility
- Use with caution

Recommended approach:

```
# Production code - exact pinning
pandas==2.3.3

# Library development - compatible release
pandas~=2.3.3

# Quick experiments - minimum version
pandas>=2.3.0
```

Installing from requirements.txt

```
# Activate virtual environment first!
source .venv/bin/activate

# Install all requirements
pip install -r requirements.txt

# Upgrade pip first (recommended)
pip install --upgrade pip
pip install -r requirements.txt

# Force reinstall if needed
pip install --force-reinstall -r requirements.txt
```

Best Practices for requirements.txt

✅ **DO**:

- Use hand-curated approach
- Group packages logically with comments
- Pin exact versions for production
- Update when adding new packages
- Test installation on clean environment

❌ **DON'T**:

- Use `pip freeze` blindly
- Include OS-specific packages
- Commit without testing
- Use vague version specifiers in production

Part 4: Command-Line Interfaces with argparse

Why CLI Interfaces Matter

Before CLI (hard-coded values):

```
# train.py
def main():
    data_path = "data/processed/train.npy"  # Hard-coded!
    model_type = "random_forest"            # Hard-coded!
    tune = False                             # Hard-coded!

    # To change: Edit the file, save, run again
```

After CLI (argparse):

```
# Flexible usage from command line
python -m src.train --data data/processed/train.npy --model random_forest
--tune
python -m src.train --data data/processed/train.npy --model
logistic_regression
python -m src.train --data other_data.npy --model gradient_boosting --tune
```

Benefits:

- ✅ No code changes needed for different runs
- ✅ Easy to automate (scripts, CI/CD)
- ✅ Standard interface everyone understands
- ✅ Self-documenting with `--help`
- ✅ Type checking and validation

Introduction to argparse

Python's `argparse` module is the standard library tool for creating CLI interfaces.

Basic example:

```
import argparse

def main():
    # Create parser
    parser = argparse.ArgumentParser(
        description='Train a machine learning model'
    )

    # Add arguments
    parser.add_argument('--data', type=str, required=True,
                        help='Path to training data')
    parser.add_argument('--model', type=str, default='random_forest',
                        help='Model type to train')
    parser.add_argument('--tune', action='store_true',
                        help='Enable hyperparameter tuning')

    # Parse arguments
```

```
    args = parser.parse_args()

    # Use arguments
    print(f"Data: {args.data}")
    print(f"Model: {args.model}")
    print(f"Tune: {args.tune}")

if __name__ == '__main__':
    main()
```

Usage:

```
# Get help
python train.py --help

# Run with arguments
python train.py --data data/train.npy --model random_forest
python train.py --data data/train.npy --model logistic_regression --tune
```

Argument Types

1. Required arguments:

```
parser.add_argument('--data', type=str, required=True,
                    help='Path to training data')
```

2. Optional arguments with defaults:

```
parser.add_argument('--model', type=str, default='random_forest',
                    help='Model type')
```

3. Boolean flags:

```
parser.add_argument('--tune', action='store_true',
                    help='Enable tuning')
```

- Present → True
- Absent → False

4. Choices (restricted values):

```
parser.add_argument('--model', type=str,
                    choices=['random_forest', 'logistic_regression',
```

```
'decision_tree'],
                    help='Model type')
```

5. Multiple values:

```
parser.add_argument('--features', nargs='+', type=str,
                    help='Feature columns')
```

Complete Training CLI Example

```
# src/train.py
import argparse
import numpy as np
from datetime import datetime
from pathlib import Path

def main():
    parser = argparse.ArgumentParser(
        description='Train machine learning models for churn prediction',
        formatter_class=argparse.RawDescriptionHelpFormatter,
        epilog='''
Examples:
  # Train Random Forest with default parameters
  python -m src.train --data data/processed/preprocessed_data.npy --model
random_forest

  # Train with hyperparameter tuning
  python -m src.train --data data/processed/preprocessed_data.npy --model
random_forest --tune

  # Train all models
  python -m src.train --data data/processed/preprocessed_data.npy --model
all
        '''
    )

    # Required arguments
    parser.add_argument('--data', type=str, required=True,
                        help='Path to preprocessed training data (.npy
file)')

    # Optional arguments
    parser.add_argument('--model', type=str,
                        default='random_forest',
                        choices=['logistic_regression', 'random_forest',
                                 'decision_tree', 'adaboost',
                                 'gradient_boosting', 'voting_classifier',
'all'],
                        help='Model type to train (default:
```

```
random_forest)')

    parser.add_argument('--tune', action='store_true',
                        help='Enable hyperparameter tuning with
GridSearchCV')

    parser.add_argument('--output-dir', type=str, default='models',
                        help='Directory to save trained models (default:
models)')

    # Parse arguments
    args = parser.parse_args()

    # Load data
    print(f"\n{'='*60}")
    print(f"Training Configuration")
    print(f"{'='*60}")
    print(f"Data: {args.data}")
    print(f"Model: {args.model}")
    print(f"Hyperparameter Tuning: {args.tune}")
    print(f"Output Directory: {args.output_dir}")
    print(f"{'='*60}\n")

    # Load data
    data = np.load(args.data, allow_pickle=True).item()
    X_train = data['X_train']
    y_train = data['y_train']

    print(f"Loaded data: X_train shape = {X_train.shape}, y_train shape =
{y_train.shape}")

    # Train model(s)
    if args.model == 'all':
        # Train all models
        models = ['logistic_regression', 'random_forest', 'decision_tree',
                  'adaboost', 'gradient_boosting', 'voting_classifier']
        for model_type in models:
            train_and_save_model(model_type, X_train, y_train, args.tune,
args.output_dir)
    else:
        train_and_save_model(args.model, X_train, y_train, args.tune,
args.output_dir)

    print(f"\n{'='*60}")
    print("Training Complete!")
    print(f"{'='*60}\n")

def train_and_save_model(model_type, X_train, y_train, tune, output_dir):
    """Train and save a single model."""
    print(f"\nTraining {model_type}...")
    start_time = datetime.now()

    # Train model (import your training functions)
    from src.train import (train_logistic_regression, train_random_forest,
```

```
                            train_decision_tree, train_adaboost,
                            train_gradient_boosting,
train_voting_classifier)

    train_funcs = {
        'logistic_regression': train_logistic_regression,
        'random_forest': train_random_forest,
        'decision_tree': train_decision_tree,
        'adaboost': train_adaboost,
        'gradient_boosting': train_gradient_boosting,
        'voting_classifier': train_voting_classifier
    }

    model = train_funcs[model_type](X_train, y_train,
tune_hyperparameters=tune)

    # Save model
    Path(output_dir).mkdir(parents=True, exist_ok=True)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    model_path = f"{output_dir}/{model_type}_{timestamp}.pkl"

    import joblib
    joblib.dump(model, model_path)

    elapsed = datetime.now() - start_time
    print(f"✓ Model saved: {model_path}")
    print(f"  Training time: {elapsed.total_seconds():.2f}s")

if __name__ == '__main__':
    main()
```

CLI Best Practices

✅ **DO**:

- Provide clear, descriptive help messages
- Use sensible defaults
- Include usage examples in epilog
- Validate input files/paths
- Print informative output
- Use `--help` flag

❌ **DON'T**:

- Make everything required (use defaults)
- Use cryptic argument names
- Skip help messages
- Forget to validate inputs
- Run silently without output

Part 5: Code Enhancements

Hyperparameter Tuning with GridSearchCV

What is Hyperparameter Tuning?

Hyperparameters are settings you choose before training (e.g., number of trees, learning rate). Unlike model parameters (learned during training), hyperparameters significantly affect model performance.

Manual tuning:

```
# Try different values manually
model1 = RandomForestClassifier(n_estimators=50, max_depth=5)
model2 = RandomForestClassifier(n_estimators=100, max_depth=10)
model3 = RandomForestClassifier(n_estimators=200, max_depth=15)
# ... tedious and time-consuming
```

GridSearchCV (automated):

```
from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

# Define parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Create model
model = RandomForestClassifier(random_state=42)

# GridSearchCV tries all combinations with cross-validation
grid_search = GridSearchCV(
    estimator=model,
    param_grid=param_grid,
    cv=5,                    # 5-fold cross-validation
    scoring='accuracy',
    n_jobs=-1,               # Use all CPU cores
    verbose=1
)

# Fit on data
grid_search.fit(X_train, y_train)

# Best model
best_model = grid_search.best_estimator_
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

```
print(f"Best parameters: {best_params}")
print(f"Best CV score: {best_score:.4f}")
```

Implementation in train.py:

```
def train_random_forest(X_train, y_train, tune_hyperparameters=False):
    """
    Train Random Forest Classifier.

    Args:
        X_train: Training features
        y_train: Training labels
        tune_hyperparameters: If True, use GridSearchCV

    Returns:
        Trained model
    """
    if tune_hyperparameters:
        print("Training with hyperparameter tuning (this may take a
while)...")

        param_grid = {
            'n_estimators': [50, 100, 200],
            'max_depth': [10, 20, None],
            'min_samples_split': [2, 5],
            'min_samples_leaf': [1, 2]
        }

        model = RandomForestClassifier(random_state=42)
        grid_search = GridSearchCV(
            estimator=model,
            param_grid=param_grid,
            cv=5,
            scoring='accuracy',
            n_jobs=-1,
            verbose=1
        )

        grid_search.fit(X_train, y_train)

        print(f"Best parameters: {grid_search.best_params_}")
        print(f"Best CV score: {grid_search.best_score_:.4f}")

        return grid_search.best_estimator_

    else:
        print("Training with default hyperparameters...")
        model = RandomForestClassifier(
            n_estimators=100,
            max_depth=10,
            random_state=42
```

```
        )
        model.fit(X_train, y_train)
        return model
```

Usage:

```
# Fast: default hyperparameters (~30 seconds)
python -m src.train --data data/processed/preprocessed_data.npy --model
random_forest

# Slow but better: tuned hyperparameters (~5-10 minutes)
python -m src.train --data data/processed/preprocessed_data.npy --model
random_forest --tune
```

Scikit-learn Pipelines for Preprocessing

Problem with manual preprocessing:

```
# Training
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
model.fit(X_train_scaled, y_train)
joblib.dump(scaler, 'scaler.pkl')  # Save separately!
joblib.dump(model, 'model.pkl')    # Save separately!

# Prediction
scaler = joblib.load('scaler.pkl')  # Load both!
model = joblib.load('model.pkl')
X_new_scaled = scaler.transform(X_new)  # Don't forget to scale!
predictions = model.predict(X_new_scaled)
```

Issues:

- ❌ Easy to forget preprocessing step
- ❌ Two files to manage
- ❌ Can apply wrong scaler
- ❌ Hard to ensure consistency

Solution: Scikit-learn Pipeline

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.ensemble import RandomForestClassifier

# Create pipeline
pipeline = Pipeline([
    ('scaler', StandardScaler()),
```

```
    ('classifier', RandomForestClassifier())
])

# Train (preprocessing happens automatically)
pipeline.fit(X_train, y_train)

# Save (saves entire pipeline as one object)
joblib.dump(pipeline, 'model_pipeline.pkl')

# Predict (preprocessing happens automatically)
pipeline = joblib.load('model_pipeline.pkl')
predictions = pipeline.predict(X_new)  # Scaling applied automatically!
```

Benefits:

- ✅ Preprocessing and model bundled together
- ✅ One file to save/load
- ✅ Automatic preprocessing during prediction
- ✅ Prevents preprocessing errors
- ✅ Cleaner, more maintainable code

Advanced: ColumnTransformer for Different Feature Types

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder

# Define feature groups
numerical_features = ['tenure', 'MonthlyCharges', 'TotalCharges']
categorical_features = ['gender', 'Contract', 'PaymentMethod']

# Create preprocessing pipeline
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numerical_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'),
categorical_features)
    ]
)

# Complete pipeline
pipeline = Pipeline([
    ('preprocessor', preprocessor),
    ('classifier', RandomForestClassifier())
])

# Use it
pipeline.fit(X_train, y_train)
predictions = pipeline.predict(X_test)
```

Implementation in preprocess.py:

```python
def create_preprocessing_pipeline(numerical_features,
categorical_features):
    """
    Create sklearn preprocessing pipeline.

    Args:
        numerical_features: List of numerical feature names
        categorical_features: List of categorical feature names

    Returns:
        ColumnTransformer pipeline
    """
    from sklearn.compose import ColumnTransformer
    from sklearn.preprocessing import StandardScaler, OneHotEncoder

    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(), numerical_features),
            ('cat', OneHotEncoder(drop='first', handle_unknown='ignore'),
             categorical_features)
        ],
        remainder='passthrough'  # Keep other columns as-is
    )

    return preprocessor

def main():
    # ... load data ...

    # Define feature groups
    numerical_features = ['tenure', 'MonthlyCharges', 'TotalCharges',
'SeniorCitizen']
    categorical_features = ['gender', 'Partner', 'Dependents',
'PhoneService',
                            'MultipleLines', 'InternetService',
'OnlineSecurity',
                            'OnlineBackup', 'DeviceProtection',
'TechSupport',
                            'StreamingTV', 'StreamingMovies', 'Contract',
                            'PaperlessBilling', 'PaymentMethod']

    # Create pipeline
    pipeline = create_preprocessing_pipeline(numerical_features,
categorical_features)

    # Fit and transform
    X_train_transformed = pipeline.fit_transform(X_train)
    X_test_transformed = pipeline.transform(X_test)

    # Save pipeline
    joblib.dump(pipeline, 'data/processed/preprocessing_pipeline.pkl')
```

```
    # Save data
    data = {
        'X_train': X_train_transformed,
        'X_test': X_test_transformed,
        'y_train': y_train,
        'y_test': y_test
    }
    np.save('data/processed/preprocessed_data.npy', data)
```

Import Organization (PEP 8)

PEP 8 is Python's style guide. Proper import organization makes code more readable and professional.

Bad (disorganized):

```
from sklearn.ensemble import RandomForestClassifier
import pandas as pd
import sys
from datetime import datetime
import numpy as np
from pathlib import Path
import argparse
from sklearn.model_selection import train_test_split
import os
```

Good (PEP 8 compliant):

```
# Standard library imports
import argparse
import os
import sys
from datetime import datetime
from pathlib import Path

# Third-party imports
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split

# Local imports
from src.utils.config import Config
from src.utils.helpers import load_data, save_model
```

PEP 8 Import Order:

1. **Standard library** (built into Python)
2. **Third-party packages** (installed via pip)

3. **Local application** (your own modules)

Blank lines:

- One blank line between groups
- Alphabetical order within groups (optional but recommended)

Why it matters:

- ✅ Immediately see what external dependencies exist
- ✅ Easier to identify missing imports
- ✅ Professional, maintainable code
- ✅ Follows Python community standards
- ✅ Better for code reviews

Part 6: Configuration Management with YAML

Why Use Configuration Files?

Problem: Hard-coded values scattered everywhere

```
# train.py
model = RandomForestClassifier(n_estimators=100, max_depth=10,
random_state=42)
test_size = 0.2
batch_size = 32

# preprocess.py
scaler_type = 'standard'
handle_missing = 'mean'

# predict.py
threshold = 0.5
```

To change anything: Edit multiple files, risk introducing bugs

Solution: Centralized Configuration

```
# configs/train_config.yaml
model:
  type: random_forest
  params:
    n_estimators: 100
    max_depth: 10
    random_state: 42

training:
  test_size: 0.2
  batch_size: 32
```

```yaml
preprocessing:
  scaler: standard
  missing_strategy: mean

prediction:
  threshold: 0.5
```

To change anything: Edit one YAML file, no code changes needed!

YAML Syntax Basics

YAML (YAML Ain't Markup Language) is a human-readable data format.

Key-value pairs:

```yaml
name: John Doe
age: 30
active: true
```

Nested structures:

```yaml
person:
  name: John Doe
  age: 30
  address:
    street: 123 Main St
    city: New York
```

Lists:

```yaml
fruits:
  - apple
  - banana
  - orange

# Or inline
colors: [red, green, blue]
```

Comments:

```yaml
# This is a comment
name: John  # Inline comment
```

Data types:

```
string: "Hello"
integer: 42
float: 3.14
boolean: true
null_value: null
```

Example Configuration Files

1. Main Configuration (`configs/config.yaml`):

```
# Project Configuration

# Project metadata
project:
  name: cmpt2500f25-project-tutorial
  version: 2.0.0
  description: Tutorial project for CMPT 2500: Customer churn prediction
for telecommunications

# Paths
paths:
  data:
    raw: data/raw/
    processed: data/processed/
    external: data/external/
  models: models/
  outputs: outputs/
  logs: logs/

# Random seed for reproducibility
random_state: 42

# Logging
logging:
  level: INFO  # DEBUG, INFO, WARNING, ERROR, CRITICAL
  format: "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
  file: logs/app.log
```

2. Training Configuration (`configs/train_config.yaml`):

```
# Training Configuration

# Model selection
model:
  type: random_forest  # Options: logistic_regression, random_forest,
decision_tree, etc.
```

```yaml
  random_state: 42

# Model-specific parameters
params:
  logistic_regression:
    max_iter: 1000
    solver: lbfgs
    C: 1.0

  random_forest:
    n_estimators: 100
    max_depth: 10
    min_samples_split: 2
    min_samples_leaf: 1

  decision_tree:
    max_depth: 10
    min_samples_split: 2

  gradient_boosting:
    n_estimators: 100
    learning_rate: 0.1
    max_depth: 5

# Training settings
training:
  test_size: 0.2
  stratify: true  # Stratify by target
  tune: false  # Enable hyperparameter tuning

# Hyperparameter tuning (used when tune=true)
tuning:
  cv: 5  # Cross-validation folds
  scoring: accuracy
  n_jobs: -1  # Use all CPU cores
  verbose: 1

  # Parameter grids for tuning
  param_grids:
    random_forest:
      n_estimators: [50, 100, 200]
      max_depth: [10, 20, None]
      min_samples_split: [2, 5, 10]
      min_samples_leaf: [1, 2, 4]

    gradient_boosting:
      n_estimators: [50, 100, 200]
      learning_rate: [0.01, 0.1, 0.2]
      max_depth: [3, 5, 7]

# Model evaluation
evaluation:
  metrics:
    - accuracy
```

```
    - precision
    - recall
    - f1_score
    - roc_auc
  save_confusion_matrix: true
  save_classification_report: true
```

3. Preprocessing Configuration (`configs/preprocess_config.yaml`):

```
# Preprocessing Configuration

# Data source
data:
  filename: WA_Fn-UseC_-Telco-Customer-Churn.csv
  target_column: Churn
  id_column: customerID

# Missing value handling
missing_values:
  strategy: drop  # Options: drop, mean, median, mode, forward_fill,
backward_fill
  threshold: 0.5  # Drop columns with >50% missing values

# Feature scaling
scaling:
  method: standard  # Options: standard, minmax, robust, none
  with_mean: true
  with_std: true

# Feature columns
features:
  categorical:
    - gender
    - Partner
    - Dependents
    - PhoneService
    - MultipleLines
    - InternetService
    - OnlineSecurity
    - OnlineBackup
    - DeviceProtection
    - TechSupport
    - StreamingTV
    - StreamingMovies
    - Contract
    - PaperlessBilling
    - PaymentMethod

  numerical:
    - SeniorCitizen
    - tenure
    - MonthlyCharges
```

```
    - TotalCharges

# Train-test split
split:
  test_size: 0.2
  random_state: 42
  stratify: true  # Stratify by target

# Pipeline options
pipeline:
  use_sklearn_pipeline: true  # Recommended: true
  save_artifacts: true  # Save pipeline and encoders
```

4. Prediction Configuration (`configs/predict_config.yaml`):

```
# Prediction Configuration

# Model
model:
  path: models/random_forest_20241027_143022.pkl
  type: random_forest

# Preprocessing
preprocessing:
  pipeline_path: data/processed/preprocessing_pipeline.pkl
  label_encoder_path: data/processed/label_encoder.pkl

# Input data
input:
  format: csv  # Options: csv, json, numpy
  path: data/new_data.csv

# Output
output:
  format: csv  # Options: csv, json, numpy
  path: predictions/predictions.csv
  include_probabilities: true
  include_confidence: true

# Batch processing
batch:
  enabled: true
  batch_size: 1000  # Process 1000 samples at a time
```

Loading YAML in Python

Install PyYAML:

```
pip install PyYAML
```

Load configuration:

```
import yaml

def load_config(config_path):
    """
    Load YAML configuration file.

    Args:
        config_path: Path to YAML file

    Returns:
        Dictionary with configuration
    """
    with open(config_path, 'r') as f:
        config = yaml.safe_load(f)
    return config

# Usage
config = load_config('configs/train_config.yaml')

# Access nested values
random_state = config['model']['random_state']
n_estimators = config['params']['random_forest']['n_estimators']
test_size = config['training']['test_size']

print(f"Random state: {random_state}")
print(f"N estimators: {n_estimators}")
print(f"Test size: {test_size}")
```

Using Configuration in Code

Example: Training with YAML config

```
# train.py
import yaml
from src.train import train_random_forest, save_model

def main():
    # Load configuration
    with open('configs/train_config.yaml', 'r') as f:
        config = yaml.safe_load(f)

    # Extract settings
    model_type = config['model']['type']
    model_params = config['params'][model_type]
    random_state = config['model']['random_state']
    tune = config['training']['tune']
```

```
    # Load data
    data_path = config['paths']['data']['processed'] + 'train_data.npy'
    data = np.load(data_path, allow_pickle=True).item()
    X_train = data['X_train']
    y_train = data['y_train']

    # Train model with config parameters
    model = train_random_forest(
        X_train,
        y_train,
        tune_hyperparameters=tune,
        **model_params
    )

    # Save model
    output_dir = config['paths']['models']
    save_model(model, model_type, output_dir)

    print(f"Model trained and saved using config: {config_path}")

if __name__ == '__main__':
    main()
```

Combining CLI and YAML

Best practice: Use YAML for defaults, CLI for overrides

```
def main():
    parser = argparse.ArgumentParser()

    parser.add_argument('--config', type=str,
                        default='configs/train_config.yaml',
                        help='Path to config file')
    parser.add_argument('--tune', action='store_true',
                        help='Override config: enable tuning')
    parser.add_argument('--model', type=str,
                        help='Override config: model type')

    args = parser.parse_args()

    # Load YAML config
    with open(args.config, 'r') as f:
        config = yaml.safe_load(f)

    # Override with CLI arguments if provided
    if args.tune:
        config['training']['tune'] = True
    if args.model:
        config['model']['type'] = args.model
```

```
    # Use merged configuration
    # ...
```

Usage:

```
# Use defaults from YAML
python train.py

# Override specific settings
python train.py --tune
python train.py --model gradient_boosting --tune
python train.py --config configs/experimental_config.yaml
```

Multiple Configurations for Different Scenarios

Create different configs for different use cases:

```
configs/
├── train_config.yaml          # Default training
├── train_config_fast.yaml     # Quick experiments (no tuning)
├── train_config_production.yaml  # Production (with tuning)
├── preprocess_config.yaml     # Default preprocessing
└── predict_config.yaml        # Prediction settings
```

Example usage:

```
# Fast experimentation
python train.py --config configs/train_config_fast.yaml

# Production training
python train.py --config configs/train_config_production.yaml
```

Benefits of YAML Configuration

Aspect	Hard-coded / config.py	YAML Configuration
Readability	❌ Python syntax	✅ Plain text
Comments	✅ Yes	✅ Yes, inline
Nested Data	⚠ Verbose	✅ Clean
Hot Reload	❌ Restart needed	✅ Just reload file
Non-programmers	❌ Hard to edit	✅ Easy to edit
Version Control	✅ Yes	✅ Yes

Aspect	Hard-coded / config.py	YAML Configuration
Multiple Configs	❌ Complex	✅ Easy

Summary of Parts 1-6

Congratulations! You've now:

✅ **Understood computational environments** and why they matter ✅ **Created virtual environments** for isolated Python installations ✅ **Managed dependencies** with `requirements.txt` ✅ **Built CLI interfaces** using argparse ✅ **Added hyperparameter tuning** to improve models ✅ **Implemented scikit-learn pipelines** for better preprocessing ✅ **Organized imports** following PEP 8 style ✅ **Created YAML configurations** for flexible settings

Your Updated Project Structure

```
your-project/
├── .venv/                          # Virtual environment (not in Git)
├── configs/                        # YAML configuration files
│   ├── train_config.yaml
│   ├── preprocess_config.yaml
│   └── predict_config.yaml
├── data/
│   ├── raw/
│   └── processed/
│       ├── preprocessed_data.npy
│       ├── preprocessing_pipeline.pkl  # NEW: Saved pipeline
│       └── label_encoder.pkl           # NEW: Saved encoder
├── src/
│   ├── preprocess.py               # UPDATED: With sklearn pipelines & CLI
│   ├── train.py                    # UPDATED: With tuning & CLI
│   ├── predict.py                  # UPDATED: With CLI
│   ├── evaluate.py                 # UPDATED: With CLI
│   └── utils/
│       └── config.py
├── models/                         # Saved models
├── requirements.txt                # NEW: Dependencies
└── README.md
```

Quick Command Reference

```
# Virtual Environment
python -m venv .venv
source .venv/bin/activate  # Mac/Linux
.venv\Scripts\activate     # Windows

# Install Dependencies
pip install -r requirements.txt
```

```
# Preprocessing (with pipeline)
python -m src.preprocess --input data/raw/data.csv

# Training (with tuning)
python -m src.train --data data/processed/preprocessed_data.npy --model
random_forest --tune

# Prediction
python -m src.predict --model models/model.pkl --data
data/processed/preprocessed_data.npy

# Evaluation
python -m src.evaluate --model models/model.pkl --data
data/processed/preprocessed_data.npy
```

Part 7: Data Version Control with DVC

Why Version Control Data?

The Problem:

```
project/
├── data/
│   ├── train_data_v1.csv
│   ├── train_data_v2.csv
│   ├── train_data_v2_final.csv
│   ├── train_data_v2_final_ACTUALLY_FINAL.csv
│   └── train_data_v2_final_use_this_one.csv  # 😱
```

Sound familiar? Data versioning is hard:

- ❌ Large files don't belong in Git
- ❌ Manual versioning is error-prone
- ❌ Hard to track which data produced which model
- ❌ Collaboration becomes messy
- ❌ Can't easily rollback to previous versions

The Solution: DVC (Data Version Control)

DVC is like Git, but for data:

- ✅ Version control for large files
- ✅ Lightweight metadata in Git
- ✅ Data stored in cloud (S3, Google Drive, DagsHub)
- ✅ Track data-model relationships
- ✅ Easy collaboration
- ✅ Reproducible pipelines

How DVC Works:

```
Git Repository (lightweight):
├── data/
│   └── raw.dvc             # Metadata file (small, in Git)
├── models/
│   └── model.pkl.dvc       # Metadata file (small, in Git)
└── .dvc/
    └── config              # DVC configuration (in Git)

DVC Remote Storage (cloud):
└── Large actual files:
    ├── data/raw/train.csv          # Actual data (not in Git)
    └── models/model.pkl            # Actual model (not in Git)
```

Git tracks: Code + DVC metadata files (`.dvc` files)
DVC tracks: Actual data + models (stored in cloud)

Part 7.1: Installing and Initializing DVC

Install DVC

```
# Activate your virtual environment
source .venv/bin/activate

# Install DVC
pip install dvc

# Verify installation
dvc version
```

Expected output:

```
DVC version: 3.63.0 (pip)
```

Initialize DVC in Your Project

```
# Navigate to project root
cd /path/to/your/project

# Initialize DVC
dvc init
```

Expected output:

```
Initialized DVC repository.

You can now commit the changes to git.

+---------------------------------------------------------------------+
|                                                                     |
|        DVC has enabled anonymous aggregate usage analytics.         |
|     Read the analytics documentation (and how to opt-out) here:     |
|             <https://dvc.org/doc/user-guide/analytics>              |
|                                                                     |
+---------------------------------------------------------------------+

What's next?
------------
- Check out the documentation: <https://dvc.org/doc>
- Get help and share ideas: <https://dvc.org/chat>
- Star us on GitHub: <https://github.com/iterative/dvc>
```

What DVC Created

```
# Check what was created
ls -la .dvc/
```

You'll see:

```
.dvc/
├── .gitignore          # Ignores DVC cache
├── config              # DVC configuration
└── tmp/                # Temporary files
```

```
# Check Git status
git status
```

Expected:

```
Changes to be committed:
  new file:   .dvc/.gitignore
  new file:   .dvc/config
  new file:   .dvcignore
  modified:   .gitignore
```

DVC automatically:

- Created `.dvc/` directory for metadata
- Created `.dvc/config` for settings
- Modified `.gitignore` to ignore DVC cache
- Staged files for Git commit

Commit DVC Initialization

```
git add .dvc .dvcignore .gitignore
git commit -m "chore: Initialize DVC for data version control"
git push
```

Part 7.2: Tracking Data with DVC

Understanding the Transition

Your data is currently tracked by Git:

```
# See what's in Git now
git ls-files data/
```

Output:

```
data/raw/WA_Fn-UseC_-Telco-Customer-Churn.csv
data/processed/preprocessed_data.npy
data/processed/preprocessing_pipeline.pkl
data/processed/label_encoder.pkl
```

Goal: Move these large files to DVC tracking, keep only metadata in Git.

Remove Data from Git Tracking

```
# Remove raw data from Git (but keep files on disk)
git rm -r --cached data/raw

# Remove processed data from Git (but keep files on disk)
git rm -r --cached data/processed
```

Note: `--cached` flag means "remove from Git tracking but keep the actual files on your disk."

Verify files still exist:

```
ls -lh data/raw/
ls -lh data/processed/
# Files should still be there!
```

Add Data to DVC Tracking

```
# Track raw data with DVC
dvc add data/raw
```

Expected output:

```
100% Adding...|████████████████████████████████████████|1/1 [00:XX, XX file/s]

To track the changes with git, run:

        git add data/raw.dvc data/.gitignore
```

What happened?

- DVC created `data/raw.dvc` (metadata file)
- DVC created/updated `data/.gitignore` (to ignore actual data)
- DVC moved actual data to `.dvc/cache/` (local cache)
- Actual files still accessible at original location (DVC creates links)

```
# Track processed data with DVC
dvc add data/processed
```

Expected output:

```
100% Adding...|████████████████████████████████████████|1/1 [00:XX, XX file/s]

To track the changes with git, run:

        git add data/processed.dvc data/.gitignore
```

Examine DVC Metadata Files

```
# View the metadata file
cat data/raw.dvc
```

Expected output:

```
outs:
- md5: 063d451250fb0faa73bc60935e759442.dir
  size: 977501
  nfiles: 1
  hash: md5
  path: raw
```

This file contains:

- **md5**: Hash of directory contents (for tracking changes)
- **size**: Total size in bytes
- **nfiles**: Number of files
- **path**: Path to data directory

```
# Check the .gitignore DVC created
cat data/.gitignore
```

Expected output:

```
/raw
/processed
```

This tells Git to ignore the actual data directories (since DVC is now managing them).

Add DVC Files to Git

```
# Add DVC metadata files to Git
git add data/raw.dvc data/processed.dvc data/.gitignore

# Check status
git status
```

Expected:

```
Changes to be committed:
  new file:   data/.gitignore
  new file:   data/processed.dvc
  new file:   data/raw.dvc
  deleted:    data/raw/WA_Fn-UseC_-Telco-Customer-Churn.csv
  deleted:    data/processed/...
```

Commit Changes

```
git commit -m "feat: Track data with DVC instead of Git

- Remove large data files from Git tracking
- Add data/raw/ to DVC (977KB)
- Add data/processed/ to DVC (1.3MB)
- DVC metadata files tracked in Git

Data now version controlled with DVC, not Git."

git push
```

Part 7.3: Setting Up Remote Storage

Your data is now tracked by DVC locally, but to collaborate or backup, you need **remote storage** (like GitHub for code, but for data).

Remote Storage Options:

1. **DagsHub** (Recommended) ⭐
    - Purpose-built for ML projects
    - Free tier includes DVC + MLflow hosting
    - S3-compatible (industry standard)
    - Easy setup
    - Web UI to browse data
2. **Google Drive** (Alternative)
    - Free 15GB storage
    - Familiar interface
    - Good for small projects
    - **Limitation**: OAuth issues in cloud environments (CodeSpaces)
3. **Amazon S3** (Production)
    - Industry standard
    - Highly scalable
    - Pay-as-you-go
    - Best for production
4. **Others**:
    - Google Cloud Storage
    - Azure Blob Storage
    - SSH/SFTP servers

For this lab, we'll use DagsHub (recommended).

Part 7.4: DagsHub Setup (Recommended)

Why DagsHub?

- ✅ **Free for students/educators**
- ✅ **DVC + MLflow in one place** (we'll use MLflow next!)
- ✅ **S3-compatible** (same as Amazon S3 - the most common storage in industry)
- ✅ **Works in all environments** (local, cloud, CodeSpaces)
- ✅ **Web UI** to browse data versions
- ✅ **Git integration** (syncs with GitHub)

Step 1: Create DagsHub Account

1. Go to https://dagshub.com/
2. Click **"Sign Up"** or **"Sign in with GitHub"** (recommended - uses your GitHub account)
3. Verify your email if prompted

Step 2: Install DagsHub and DVC-S3 Package

```
# Install DagsHub CLI and authentication tools
pip install dagshub --upgrade
pip install dvc-s3
```

Step 3: Authenticate with DagsHub

```
# Login to DagsHub (creates authentication token)
dagshub login
```

This will:

1. Open a browser window (or provide a URL to open)
2. Ask you to authorize DagsHub CLI
3. Prompt you to select token expiration time

Important Notes:

- ⚠ **Token Expiration**: Choose a timeframe that covers your course duration (e.g., 3 months for a semester course). The token will expire after this period.
- ⚠ **New CodeSpaces Instances**: If you create a new CodeSpaces environment, you'll need to run `dagshub login` again to re-authenticate.
- ✅ **Token Storage**: The authentication token is stored locally in `~/.dagshub/config` (not in your project, not in Git).

Expected output:

```
DagsHub login successful!
Authentication token saved to ~/.dagshub/config
```

Step 4: Create DagsHub Repository

1. Click **"+ New Repository"** (top right)
2. Fill in details:
    - **Repository name**: Match your GitHub repo name
    - **Description**: "Describe your problem"
    - **Visibility**: Public or Private (your choice; private if you are asked by data provider not to share)
    - **Initialize with**: Leave all unchecked (we already have a repo)
3. Click **"Create Repository"**

You'll see an empty repository page with setup instructions.

Example (instructor's repo):

```
https://dagshub.com/ajallooe/cmpt2500f25-project-tutorial
```

Step 5: Get DagsHub Credentials

On your DagsHub repository page:

1. Look for **"Connection credentials"** box (usually bottom right)
2. Click **"Simple Data Upload"** tab
3. You'll see:
    - **Bucket name**: Your repository name
    - **Endpoint URL**: `https://dagshub.com/api/v1/repo-buckets/s3/your-username`
    - **Access Key ID**: (a token)
    - **Secret Access Key**: (same token - DagsHub uses the same value for both)
    - **Region**: `us-east-1`

Keep this page open - you'll need these credentials!

Step 6: Configure DVC Remote

Back in your terminal:

```
# Add DagsHub as DVC remote
# Replace with YOUR values from DagsHub
dvc remote add origin s3://dvc
dvc remote modify origin endpointurl https://dagshub.com/your-
username/your-project-name.s3

# Set as default remote
dvc remote default origin
```

Example (with actual values):

```
dvc remote add origin s3://dvc
dvc remote modify origin endpointurl
https://dagshub.com/ajallooe/cmpt2500f25-project-tutorial.s3
dvc remote default origin
```

Step 7: Add Credentials (Stored Locally Only)

```
# Add credentials (replace YOUR_TOKEN with actual token from DagsHub)
dvc remote modify origin --local access_key_id YOUR_TOKEN
dvc remote modify origin --local secret_access_key YOUR_TOKEN
```

Important: The `--local` flag stores credentials in `.dvc/config.local`, which is automatically ignored by Git. Your credentials stay secure on your machine only!

Step 8: Verify Configuration

```
# Check main config (will be committed to Git)
cat .dvc/config
```

Expected output:

```
[core]
    remote = origin
['remote "origin"']
    url = s3://dvc
    endpointurl = https://dagshub.com/your-username/your-repo.s3
```

```
# Check local config (NOT committed - has credentials)
cat .dvc/config.local
```

Expected output:

```
['remote "origin"']
    access_key_id = YOUR_TOKEN
    secret_access_key = YOUR_TOKEN
```

```
# Verify remote is set
dvc remote list
```

Expected output:

```
origin  s3://dvc    (default)
```

Step 9: Commit Remote Configuration

```
# Add config to Git (NOT config.local - that's gitignored)
git add .dvc/config
git commit -m "chore: Configure DagsHub as DVC remote storage

- Add DagsHub S3-compatible remote
- Set as default remote
- Credentials stored locally (not committed)"

git push
```

Step 10: Push Data to DagsHub

```
# Push data to remote storage
dvc push
```

Expected output:

```
Collecting
Pushing
2 files pushed
```

This uploads:

- `data/raw/` (955KB)
- `data/processed/` (1.3MB)

First push might take a minute depending on your internet speed.

Step 11: Verify Upload

```
# Check DVC status
dvc status -c
```

Expected output:

```
Cache and remote 'origin' are in sync.
```

This confirms your local data and remote data match!

Optional: Check DagsHub Web UI

1. Go to your DagsHub repository
2. Click **"Files"** tab
3. Navigate to `data/` directory
4. You should see `.dvc` metadata files (Git tracks these)
5. The actual data is in DagsHub's storage (not visible in files, but in Storage tab)

Note: DagsHub UI can take a few minutes to sync. The important thing is that `dvc status -c` shows everything is in sync.

Part 7.5: Testing DVC Workflow

Let's test that DVC actually works by simulating a fresh clone:

Simulate Fresh Clone

```
# Remove local cache
rm -rf .dvc/cache

# Verify cache is gone
ls -la .dvc/
# Should NOT see cache/ directory

# Remove actual data
rm -rf data/raw/WA_Fn-UseC_-Telco-Customer-Churn.csv
rm -rf data/processed/*.pkl data/processed/*.npy

# Verify data is gone
ls -la data/raw/
ls -la data/processed/
# Should be empty
```

Pull Data from DagsHub

```
# Pull data from remote
dvc pull
```

Expected output:

```
Collecting
Fetching
2 files fetched
```

Verify Data Restored

```
# Check if data is back
ls -lh data/raw/
ls -lh data/processed/
```

Expected: All your files should be back with correct sizes!

```
data/raw/:
  WA_Fn-UseC_-Telco-Customer-Churn.csv (955KB)

data/processed/:
  preprocessed_data.npy (1.2MB)
  preprocessing_pipeline.pkl (48KB)
  label_encoder.pkl (484B)
```

Success! ✅ DVC is working correctly. You can now:

- Version control your data
- Collaborate with teammates (they just `dvc pull`)
- Rollback to previous data versions
- Track which data produced which model

Part 7.6: Google Drive Setup (Alternative)

⚠ **Important Note**: Google Drive remote has limitations in cloud environments (GitHub CodeSpaces, AWS Cloud9, etc.) due to OAuth authentication restrictions. **DagsHub is strongly recommended for cloud development.**

Google Drive works well for:

- ✅ Local development (your own computer)
- ✅ Small projects
- ✅ Personal learning

Use Google Drive only if:

- You're working on your local machine (not cloud)
- Your project is small (<15GB)

- You want to learn alternative remotes

Prerequisites for Google Drive

If you want to use Google Drive, you'll need to:

1. **Install Google Drive support**:

```
pip install dvc-gdrive
```

2. **Create a Google Drive folder**:
   - Go to https://drive.google.com
   - Create folder: `dvc-your-project-name`
   - Share with team members and instructor
   - Get the folder ID from URL (after `/folders/`)

3. **Configure DVC remote**:

```
dvc remote add -d gdrive gdrive://YOUR_FOLDER_ID
```

4. **Authenticate (local machine only)**:

```
dvc push
# This will open browser for Google OAuth
# Follow prompts to authenticate
```

OAuth Limitation in Cloud Environments

Why it doesn't work in CodeSpaces/cloud:

When you run `dvc push` with Google Drive, it tries to:

1. Open a browser window for Google authentication
2. Ask you to grant permissions to PyDrive2 (DVC's Google Drive library)
3. Get an authorization code back

The problem:

- Cloud environments (CodeSpaces, Cloud9) can't open browsers
- Google blocks "unverified apps" for security
- PyDrive2 is considered an "unverified app"

Error you'll see:

> This app is blocked This app tried to access sensitive info in your Google Account. To keep your account safe, Google blocked this access.

Workaround: Service Account (Advanced)

For Google Drive to work in cloud environments, you need to:

1. Create a Google Cloud Project
2. Enable Google Drive API
3. Create a service account
4. Generate credentials JSON file
5. Share your Drive folder with service account email
6. Configure DVC with service account JSON

This process:

- Takes 15-20 minutes to set up
- Requires Google Cloud Console access
- Is more complex than DagsHub
- **Not tested in CodeSpaces for this lab**

We provide this information for reference, but recommend DagsHub for ease of use.

Part 7.7: DVC Branch Management

Remember we created a `dvc-google-drive` branch? Let's understand the branching strategy:

Main Branch (DagsHub)

```
# Ensure you're on main
git checkout main

# Verify DagsHub remote
dvc remote list
```

Output:

```
origin  s3://dvc    (default)
```

This branch is:

- ✅ Fully working and tested
- ✅ Recommended for all students
- ✅ Works in cloud environments
- ✅ Production-ready

Google Drive Branch (Alternative)

```
# Switch to Google Drive branch (for reference)
git checkout dvc-google-drive

# Verify Google Drive remote
dvc remote list
```

Output:

```
gdrive  gdrive://FOLDER_ID    (default)
```

This branch is:

- ⚠ For local development only
- ⚠ OAuth issues in cloud environments
- ⚠ Documented but not fully tested
- ℹ️ Learning resource for alternative remotes

Don't worry about this branch for now. Stick with main (DagsHub)!

Part 7.8: DVC Workflow Summary

Daily Workflow

1. Make changes to data:

```
# Preprocess data (creates/updates files)
python -m src.preprocess --input data/raw/new_data.csv

# DVC notices files changed
dvc status
```

2. Track changes with DVC:

```
# Add changed files to DVC
dvc add data/processed

# Commit DVC metadata to Git
git add data/processed.dvc
git commit -m "feat: Update processed data with new preprocessing"
git push
```

3. Push data to remote:

```
# Upload actual data to DagsHub
dvc push
```

4. Teammates get your changes:

```
# Teammate pulls code
git pull

# Teammate pulls data
dvc pull
```

Common Commands

```
# Check what changed
dvc status

# Check if local and remote are in sync
dvc status -c

# Add/update data tracking
dvc add data/raw
dvc add data/processed

# Push data to remote
dvc push

# Pull data from remote
dvc pull

# List remotes
dvc remote list

# Check out specific data version (like git checkout for data)
git checkout <commit-hash>
dvc pull
```

Part 7.9: Updating requirements.txt

Don't forget to add DVC to your dependencies!

```
# Check if DVC is already in requirements.txt
grep dvc requirements.txt
```

If not present, add it:

```
# Add to requirements.txt

# Data Version Control (DagsHub remote - recommended)
dagshub==0.6.3
dvc-s3==3.2.2
```

Note: If using Google Drive alternative, add:

```
dvc-gdrive==3.0.1
```

Commit the update:

```
git add requirements.txt
git commit -m "chore: Add DVC dependencies to requirements.txt"
git push
```

Part 7.10: DVC Best Practices

✅ **DO**:

- Track large files (>10MB) with DVC
- Track data and trained models with DVC
- Keep small artifacts (<1MB) in Git (preprocessing_pipeline.pkl is borderline)
- Commit `.dvc` files to Git
- Use `dvc push` after `dvc add`
- Document remote setup in README

❌ **DON'T**:

- Track code with DVC (use Git for code)
- Commit `.dvc/cache/` to Git (it's gitignored for a reason)
- Commit `.dvc/config.local` to Git (contains credentials!)
- Forget to push data (`dvc push` after changes)
- Mix data versions across branches without care

Part 7.11: Troubleshooting DVC

Problem: `dvc push` fails with "Permission denied"

Solution: Check credentials:

```

```

```
cat .dvc/config.local
# Verify tokens are correct
```

Problem: `dvc pull` says "Unable to find file"

Solution: Ensure remote has the data:

```
dvc status -c
# If out of sync, check Git commit and data version match
```

Problem: "Output 'data/raw' is already tracked by SCM"

Solution: Remove from Git first:

```
git rm -r --cached data/raw
dvc add data/raw
```

Problem: Large data file accidentally in Git

Solution: Remove from history:

```
# Use BFG or git filter-branch (advanced)
# Better: Prevent by using .gitignore from start
```

Summary of Part 7: DVC

Congratulations! You've now:

✅ **Installed and initialized DVC** ✅ **Tracked data with DVC** (removed from Git) ✅ **Set up DagsHub remote** (S3-compatible storage) ✅ **Pushed data to cloud storage** ✅ **Tested pull workflow** (simulated fresh clone) ✅ **Understood Google Drive alternative** (with limitations) ✅ **Updated requirements.txt** with DVC dependencies

Updated Project Structure

```
your-project/
├── .dvc/
│   ├── .gitignore          # Ignores cache
│   ├── config              # Remote config (IN GIT)
│   ├── config.local        # Credentials (NOT in Git)
│   └── cache/              # Local data cache (NOT in Git)
├── data/
│   ├── .gitignore          # Created by DVC
```

```
│   ├── raw.dvc              # Metadata (IN GIT)
│   ├── processed.dvc        # Metadata (IN GIT)
│   ├── raw/                 # Actual data (IN DVC CACHE)
│   └── processed/           # Actual data (IN DVC CACHE)
├── src/
│   ├── preprocess.py
│   ├── train.py
│   ├── predict.py
│   └── evaluate.py
├── models/
├── requirements.txt         # Now includes: dvc, dagshub, dvc-s3
└── README.md
```

What's tracked where:

Item	Git	DVC	DagsHub
Code (`.py`)	✅ Yes	❌ No	✅ Synced from Git
Config (`.yaml`)	✅ Yes	❌ No	✅ Synced from Git
DVC metadata (`.dvc`)	✅ Yes	❌ No	✅ Synced from Git
Data files	❌ No	✅ Yes	✅ Yes (storage)
Model files	❌ No	✅ Yes	✅ Yes (storage)
Credentials	❌ No	🔒 Local only	❌ No

Part 8: Experiment Tracking with MLflow

Introduction: The Experiment Tracking Problem

Imagine you're training multiple machine learning models:

```
Monday: Random Forest, n_estimators=100, max_depth=10 → Accuracy: 79.3%
Tuesday: Random Forest, n_estimators=200, max_depth=15 → Accuracy: 80.1%
Wednesday: Random Forest, n_estimators=150, max_depth=12 → Accuracy: ???
```

Questions you can't answer:

- ❌ Which hyperparameters gave the best results?
- ❌ What was the exact configuration of Monday's model?
- ❌ How do I compare all experiments side-by-side?
- ❌ Where did I save that good model from Tuesday?
- ❌ What data version was used for each experiment?

Traditional "solutions" (all problematic):

- Spreadsheet tracking (manual, error-prone)

- Text file logging (unstructured, hard to analyze)
- Print statements (lost after terminal closes)
- Manual naming conventions (inconsistent, confusing)

What is MLflow?

MLflow is an open-source platform for managing the end-to-end machine learning lifecycle.

Created by: Databricks (2018)
Industry Adoption: Microsoft, Toyota, Netflix, Walmart
Why it's standard: Works with any ML library (scikit-learn, TensorFlow, PyTorch)

Part 8.1: Installing MLflow

Step 1: Install MLflow

```
# Activate virtual environment
source .venv/bin/activate

# Install MLflow
pip install mlflow

# Verify installation
mlflow --version
```

Step 2: Update requirements.txt

```
echo "mlflow==3.5.1" >> requirements.txt
```

Or manually add to `requirements.txt`:

```
mlflow==3.5.1
```

Commit:

```
git add requirements.txt
git commit -m "chore: Add MLflow for experiment tracking"
git push
```

Part 8.2: Understanding MLflow Tracking

MLflow Hierarchy

```
Experiment (e.g., "telecom-churn-prediction")
├── Run 1 (Random Forest, 100 trees)
│   ├── Parameters: n_estimators=100, max_depth=10
│   ├── Metrics: accuracy=0.793, f1=0.585
│   └── Artifacts: model.pkl, confusion_matrix.png
├── Run 2 (Random Forest, 200 trees)
│   ├── Parameters: n_estimators=200, max_depth=15
│   ├── Metrics: accuracy=0.801, f1=0.603
│   └── Artifacts: model.pkl, confusion_matrix.png
```

Key Concepts:

- **Experiment**: Collection of related runs
- **Run**: One execution of training code
- **Parameters**: Input settings (hyperparameters)
- **Metrics**: Output measurements (accuracy, loss)
- **Artifacts**: Output files (models, plots)
- **Tags**: Metadata for organization

Part 8.3: Updating train.py with MLflow

Step 1: Add MLflow Imports

At the top of `src/train.py`:

```python
import mlflow
import mlflow.sklearn
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score,
    f1_score, roc_auc_score, confusion_matrix
)
```

Step 2: Set MLflow Experiment

In your `main()` function:

```python
def main():
    # ... argument parsing ...

    # Set MLflow experiment
    mlflow.set_experiment("telecom-churn-prediction")

    # ... rest of code ...
```

Step 3: Add MLflow Logging to Training Functions

Update each training function to log parameters:

```
def train_random_forest(X_train, y_train, tune_hyperparameters=False,
**kwargs):
    """Train Random Forest with MLflow logging."""

    if tune_hyperparameters:
        param_grid = {...}
        mlflow.log_param("param_grid", str(param_grid))  # Log grid

        # ... GridSearchCV code ...

        # Log best parameters
        for param, value in grid_search.best_params_.items():
            mlflow.log_param(f"best_{param}", value)
        mlflow.log_metric("cv_best_score", grid_search.best_score_)
    else:
        # Log default parameters
        params = {'n_estimators': 100, 'max_depth': None}
        for key, value in params.items():
            mlflow.log_param(key, value)

    # ... rest of training ...
```

Step 4: Wrap Training in MLflow Run

In your `main()` function, wrap training:

```
def main():
    # ... setup code ...

    if args.model == 'all':
        # Train each model in its own MLflow run
        for model_name in ['random_forest', 'gradient_boosting', ...]:
            with mlflow.start_run(run_name=f"{model_name}_{timestamp}"):
                # Log tags
                mlflow.set_tag("model_type", model_name)
                mlflow.set_tag("tuning", "enabled" if args.tune else
"disabled")

                # Log parameters
                mlflow.log_param("model_type", model_name)
                mlflow.log_param("tune_hyperparameters", args.tune)

                # Train model
                model = train_model(...)
```

```
                # Evaluate and log metrics
                metrics = evaluate_model(model, X_test, y_test)
                for metric_name, value in metrics.items():
                    mlflow.log_metric(metric_name, value)

                # Log model
                mlflow.sklearn.log_model(model, "model")

                # Log local file as artifact
                mlflow.log_artifact(model_path, "local_models")
```

Step 5: Add Evaluation Function

```
def evaluate_model(model, X_test, y_test):
    """Evaluate model and return metrics."""
    yhat = model.predict(X_test)

    metrics = {
        'accuracy': accuracy_score(y_test, yhat),
        'precision': precision_score(y_test, yhat, average='binary',
pos_label='Yes'),
        'recall': recall_score(y_test, yhat, average='binary',
pos_label='Yes'),
        'f1_score': f1_score(y_test, yhat, average='binary',
pos_label='Yes')
    }

    # ROC-AUC if model supports predict_proba
    if hasattr(model, 'predict_proba'):
        y_proba = model.predict_proba(X_test)[:, 1]
        metrics['roc_auc'] = roc_auc_score(y_test == 'Yes', y_proba)

    return metrics
```

Part 8.4: Running Training with MLflow

Train a Single Model

```
python -m src.train \
    --data data/processed/preprocessed_data.npy \
    --model random_forest

# Output shows MLflow run ID:
# MLflow run ID: abc123def456...
```

Train with Hyperparameter Tuning

```
python -m src.train \
    --data data/processed/preprocessed_data.npy \
    --model random_forest \
    --tune
```

Train All Models

```
python -m src.train \
    --data data/processed/preprocessed_data.npy \
    --model all
```

Part 8.5: Accessing MLflow UI in CodeSpaces

CRITICAL: Port Forwarding Required

Why: CodeSpaces is a remote virtual machine. When you run `mlflow ui`, it starts on that remote machine's localhost, not your computer.

Step 1: Start MLflow UI

```
# Must use 0.0.0.0 to allow external connections
mlflow ui --host 0.0.0.0 --port 5000
```

Expected output:

```
[2024-10-27 10:30:00] [INFO] Listening at: http://0.0.0.0:5000
```

Leave this terminal running! Open a new terminal for other commands.

Step 2: Forward Port in CodeSpaces

In VS Code:

1. **Look at bottom panel** - Click **PORTS** tab
    - Should see: `PROBLEMS | OUTPUT | DEBUG CONSOLE | TERMINAL | PORTS`
2. **Port 5000 should appear automatically**
    - If not, click **"Forward a Port"** button (or **+** icon)
    - Enter: `5000`
    - Press Enter

3. **Open in Browser**:
    - Right-click on port 5000 row
    - Select **"Open in Browser"**

    OR

    - Hover over "Forwarded Address" column
    - Click globe icon 🌐

    OR

    - Copy the URL (like `https://username-repo-abc123-5000.app.github.dev`)
    - Paste into new browser tab

Step 3: Verify MLflow UI

You should see:

- Experiments list (left sidebar)
- "telecom-churn-prediction" experiment
- Runs table with your training runs
- Metrics columns (accuracy, f1_score, etc.)

Troubleshooting:

- "Can't reach this page" → Check mlflow ui still running
- Blank page → Try incognito/private window
- Port not listed → Manually forward port 5000
- Still not working → See troubleshooting section below

Part 8.6: Comparing Experiments

Step 1: Train Multiple Models

```
python -m src.train --data data.npy --model random_forest
python -m src.train --data data.npy --model random_forest --tune
python -m src.train --data data.npy --model gradient_boosting
python -m src.train --data data.npy --model logistic_regression
```

Step 2: Select Runs

In MLflow UI:

1. Check boxes next to 2-4 runs
2. Click **"Compare"** button

Step 3: View Comparison

Comparison page shows:

- **Parallel Coordinates Plot**: Visual comparison
- **Scatter Plot**: Parameter vs. metric correlation
- **Parameters Table**: Side-by-side parameter comparison
- **Metrics Table**: Side-by-side metric comparison

To find best model:

1. Sort metrics table by "accuracy" (click column header)
2. Note the best run ID
3. Check other metrics (F1, ROC-AUC)
4. Consider training time

Part 8.7: Loading Models from MLflow

Method 1: Load by Run ID

```
import mlflow.sklearn

# Get run ID from MLflow UI
run_id = "abc123def456..."

# Load model
model = mlflow.sklearn.load_model(f"runs:/{run_id}/model")

# Use model
predictions = model.predict(X_test)
```

Method 2: Load Best Model

```
import mlflow

# Search for best run
mlflow.set_experiment("telecom-churn-prediction")
runs = mlflow.search_runs(
    order_by=["metrics.accuracy DESC"],
    max_results=1
)

# Load best model
best_run_id = runs.iloc[0]['run_id']
model = mlflow.sklearn.load_model(f"runs:/{best_run_id}/model")

print(f"Loaded model from run: {best_run_id}")
print(f"Accuracy: {runs.iloc[0]['metrics.accuracy']}")
```

Part 8.8: MLflow Best Practices

What to Log

Always log:

- ✅ Model type and version
- ✅ All hyperparameters (including defaults)
- ✅ Performance metrics (accuracy, precision, recall, F1)
- ✅ Training time
- ✅ Random seed

Consider logging:

- Cross-validation scores
- Confusion matrix values
- Feature importance
- Data version (DVC hash)

Naming Conventions

Experiments:

```
✅ "telecom-churn-prediction"
✅ "customer-churn-v2"
❌ "experiment1"
❌ "test"
```

Run names:

```
✅ "random_forest_20241027_103045"
✅ "rf_tuned_final"
❌ "run1"
❌ "final_final_v2"
```

Organization Tips

```
# Use tags for filtering
mlflow.set_tag("model_type", "random_forest")
mlflow.set_tag("data_version", "v1.0")
mlflow.set_tag("tuning", "enabled")
mlflow.set_tag("environment", "development")
```

Part 8.9: Troubleshooting

Problem: MLflow UI Not Accessible

Symptoms: "Can't reach this page", connection refused

Solutions:

```
# 1. Check MLflow is running
# Terminal should show "Listening at: http://0.0.0.0:5000"

# 2. Restart with correct host
mlflow ui --host 0.0.0.0 --port 5000

# 3. In PORTS tab:
# - Verify port 5000 is listed
# - Right-click → "Port Visibility" → "Public"
# - Click "Open in Browser"
```

Problem: Runs Not Showing

Solutions:

1. Refresh browser (F5)
2. Check experiment name matches
3. Verify mlruns/ directory exists: `ls mlruns/`

Problem: Large mlruns/ Directory

Solution:

```
# Delete old experiments
mlflow experiments delete --experiment-id 1

# Or delete via UI (select runs → delete)
```

Part 8.10: Update .gitignore

Add to `.gitignore`:

```
# MLflow
mlruns/
mlflow-artifacts/
mlartifacts/
mlflow.db*
.mlflow/
```

Commit:

```
git add .gitignore
git commit -m "chore: Update .gitignore for MLflow"
```

Summary

You've completed MLflow integration! You can now:

✅ Track experiments automatically
✅ Compare models visually
✅ Load best models by metric
✅ Access MLflow UI in CodeSpaces
✅ Reproduce results reliably

Next Steps

View your experiments:

```
mlflow ui --host 0.0.0.0 --port 5000
# PORTS tab → Forward 5000 → Open in Browser
```

Train and compare:

```
python -m src.train --data data.npy --model all --tune
# Then compare in MLflow UI
```

Part 9: Automated Testing with pytest

Testing Overview

This is the final piece of your production-ready ML project! In this section, you'll add **automated testing** to ensure your code is reliable, maintainable, and bug-free.

Testing is a critical part of software engineering that's often overlooked in ML projects. However, in production environments, untested code is a major risk. This lab will teach you industry-standard testing practices using **pytest**, the most popular Python testing framework.

What You've Built So Far

Lab 02 has been an intensive journey:

- ✅ **Part 1-2**: Virtual environments & dependency management

- ✅ **Part 3-4**: CLI interfaces & code enhancements
- ✅ **Part 5-6**: YAML configuration & best practices
- ✅ **Part 7**: DVC for data version control
- ✅ **Part 8**: MLflow for experiment tracking
- 🎯 **Part 9**: Automated testing (You are here!)

Testing Learning Objectives

By the end of this part, you will:

1. **Understand testing fundamentals** - Why test, what to test, how to test
2. **Master pytest** - Write and run unit tests, integration tests, and parametrized tests
3. **Create test fixtures** - Reusable test data and configurations
4. **Measure code coverage** - Ensure your tests cover critical code paths
5. **Test ML pipelines** - Special considerations for testing ML code
6. **Mock external dependencies** - Test without hitting databases or APIs

Why Testing Matters

The cost of bugs in production:

```
Development:   Bug costs $1 to fix
Testing:       Bug costs $10 to fix
Production:    Bug costs $100 to fix
```

Real-world ML disasters caused by lack of testing:

- Amazon's AI recruiting tool showed bias against women (insufficient testing for fairness)
- Knight Capital lost $440 million in 45 minutes (deployment without proper testing)
- Mars Climate Orbiter crashed ($125M loss) due to unit conversion error

Testing benefits:

- ✅ Catch bugs before production
- ✅ Refactor confidently
- ✅ Document expected behavior
- ✅ Enable continuous integration/deployment
- ✅ Improve code quality

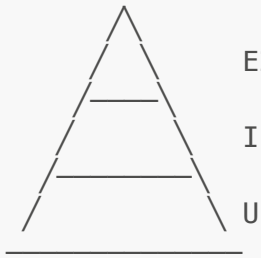
Part 9.1: Understanding Testing Fundamentals

Types of Tests

Test Type	What it Tests	Example	Speed	Coverage
Unit Tests	Individual functions	`test_load_data()` tests just `load_data()`	⚡ Fast	Narrow

Test Type	What it Tests	Example	Speed	Coverage
Integration Tests	Multiple components	`test_preprocessing_pipeline()` tests full flow	🦕 Slower	Broad
End-to-End Tests	Complete workflows	CSV → Model → Predictions	🐌 Slowest	Complete

The Testing Pyramid



Best practice: Write mostly unit tests, some integration tests, and few E2E tests.

What to Test in ML Projects

Always test:

- ✅ Data loading and validation
- ✅ Preprocessing functions
- ✅ Model training (that it completes without errors)
- ✅ Prediction functions
- ✅ Evaluation metrics
- ✅ Edge cases (empty data, missing values, wrong types)

Don't test:

- ❌ Third-party library internals (scikit-learn already tests their code)
- ❌ Exact model accuracy (can vary, test that it's in a reasonable range)
- ❌ Hyperparameter tuning results (non-deterministic, too slow)

Test-Driven Development (TDD)

TDD Cycle (optional, but recommended):

1. **Red**: Write a failing test
2. **Green**: Write minimal code to make it pass
3. **Refactor**: Improve the code while keeping tests passing

---

Part 9.2: Setting Up pytest

Install pytest

```
# Activate your virtual environment
source .venv/bin/activate

# Install pytest and pytest-cov
pip install pytest==8.4.2 pytest-cov==7.0.0

# Verify installation
pytest --version
```

Expected output:

```
pytest 8.4.2
```

Create pytest.ini Configuration

Create a `pytest.ini` file in your project root:

```
touch pytest.ini
```

Add this configuration:

```
# pytest.ini - pytest Configuration

[pytest]
# Test discovery patterns
python_files = test_*.py
python_classes = Test*
python_functions = test_*

# Directories to search for tests
testpaths = tests

# Additional command line options
addopts =
    -v
    --strict-markers
    --tb=short
    --disable-warnings
    -ra

# Markers for organizing tests
markers =
    unit: Unit tests for individual functions
    integration: Integration tests for workflows
    slow: Tests that take longer to run
    cli: Tests for command-line interfaces
```

```
# Minimum Python version
minversion = 3.8

# Output options
console_output_style = progress

# Warnings
filterwarnings =
    ignore::DeprecationWarning
```

What this does:

- `python_files = test_*.py`: Only run files starting with `test_`
- `testpaths = tests`: Look for tests in `tests/` directory
- `-v`: Verbose output
- `markers`: Custom tags for organizing tests

Create Tests Directory Structure

```
mkdir -p tests
touch tests/__init__.py
touch tests/conftest.py
```

Your project now looks like:

```
your-project/
├── tests/
│   ├── __init__.py
│   └── conftest.py
├── pytest.ini
└── src/
    ├── preprocess.py
    ├── train.py
    ├── predict.py
    └── evaluate.py
```

Part 9.3: Writing Your First Test

Create test_preprocess.py

Create `tests/test_preprocess.py`:

```
"""
Tests for src/preprocess.py module.
```

```
"""

import pandas as pd
import pytest

from src.preprocess import load_data


def test_load_data_success(tmp_path):
    """Test successful data loading from CSV."""
    # Create a temporary CSV file
    csv_file = tmp_path / "test_data.csv"
    csv_file.write_text("col1,col2\\n1,2\\n3,4\\n")

    # Load data
    df = load_data(str(csv_file))

    # Assertions
    assert isinstance(df, pd.DataFrame)
    assert len(df) == 2
    assert list(df.columns) == ['col1', 'col2']


def test_load_data_file_not_found():
    """Test error handling for missing file."""
    with pytest.raises(FileNotFoundError):
        load_data("nonexistent_file.csv")
```

Run Your First Test

```
pytest tests/test_preprocess.py -v
```

Expected output:

```
tests/test_preprocess.py::test_load_data_success PASSED           [ 50%]
tests/test_preprocess.py::test_load_data_file_not_found PASSED    [100%]

==================== 2 passed in 0.05s ====================
```

Congratulations! You just wrote and ran your first tests! 🎉

Part 9.4: Understanding Test Structure

Anatomy of a Test Function

```python
def test_function_name():              # 1. Name starts with 'test_'
    """Docstring explaining what we test."""  # 2. Documentation

    # 3. Arrange: Set up test data
    X = np.array([[1, 2], [3, 4]])
    y = np.array([0, 1])

    # 4. Act: Execute the code being tested
    model = train_model(X, y)

    # 5. Assert: Verify the results
    assert model is not None
    assert hasattr(model, 'predict')
```

The 3 A's of Testing (AAA Pattern):

1. **Arrange**: Set up test data and preconditions
2. **Act**: Execute the function/code being tested
3. **Assert**: Check that results match expectations

Common Assertions

```python
# Equality
assert x == y
assert x != y

# Identity
assert x is None
assert x is not None

# Membership
assert 'key' in dictionary
assert item in list

# Type checking
assert isinstance(obj, ClassName)

# Comparisons
assert x > y
assert x >= y
assert x < y
assert x <= y

# Boolean
assert condition
assert not condition

# Exceptions
with pytest.raises(ValueError):
    function_that_should_raise()
```

```
# Floating point (with tolerance)
assert abs(x - y) < 0.0001
# or
import numpy as np
np.testing.assert_almost_equal(x, y, decimal=4)
```

Part 9.5: Creating Test Fixtures

What are Fixtures?

Fixtures are reusable test data or configurations. Instead of creating the same test data in every test, you create it once as a fixture.

Create conftest.py with Fixtures

Edit `tests/conftest.py`:

```
"""
pytest fixtures for test suite.
"""

import numpy as np
import pandas as pd
import pytest


@pytest.fixture
def sample_data():
    """
    Create small sample dataset for testing.
    """
    data = {
        'customerID': ['C001', 'C002', 'C003'],
        'gender': ['Male', 'Female', 'Male'],
        'tenure': [12, 24, 36],
        'MonthlyCharges': [50.5, 70.25, 25.0],
        'Churn': ['No', 'No', 'Yes']
    }
    return pd.DataFrame(data)


@pytest.fixture
def processed_data():
    """
    Create preprocessed training data.
    """
    np.random.seed(42)

    X_train = np.random.randn(80, 15)
```

```
    X_test = np.random.randn(20, 15)
    y_train = np.random.choice(['Yes', 'No'], 80)
    y_test = np.random.choice(['Yes', 'No'], 20)

    return {
        'X_train': X_train,
        'X_test': X_test,
        'y_train': y_train,
        'y_test': y_test
    }


@pytest.fixture
def temp_output_dir(tmp_path):
    """
    Create temporary output directory.
    """
    output_dir = tmp_path / "outputs"
    output_dir.mkdir()
    return output_dir
```

Using Fixtures in Tests

```
def test_with_fixture(sample_data):
    """Test using the sample_data fixture."""
    # sample_data is automatically passed by pytest
    assert len(sample_data) == 3
    assert 'Churn' in sample_data.columns


def test_with_multiple_fixtures(sample_data, processed_data):
    """Test using multiple fixtures."""
    assert len(sample_data) > 0
    assert processed_data['X_train'].shape[0] == 80
```

pytest automatically:

1. Sees fixture name in function parameters
2. Calls the fixture function
3. Passes the return value to your test

Part 9.6: Testing Preprocessing Module

Create comprehensive tests for `test_preprocess.py`:

```
"""
Tests for src/preprocess.py module.
"""
```

```
import numpy as np
import pandas as pd
import pytest

from src.preprocess import (
    handle_missing_values,
    drop_unnecessary_columns,
    encode_target,
    split_features_target,
    preprocess_pipeline
)


class TestLoadData:
    """Tests for load_data function."""

    def test_load_data_success(self, tmp_path):
        """Test successful data loading."""
        from src.preprocess import load_data

        csv_file = tmp_path / "data.csv"
        csv_file.write_text("col1,col2\\nval1,val2\\n")

        df = load_data(str(csv_file))

        assert isinstance(df, pd.DataFrame)
        assert not df.empty

    def test_load_data_file_not_found(self):
        """Test error for missing file."""
        from src.preprocess import load_data

        with pytest.raises(FileNotFoundError):
            load_data("nonexistent.csv")


class TestHandleMissingValues:
    """Tests for handle_missing_values function."""

    def test_no_missing_values(self, sample_data):
        """Test with no missing values."""
        df_clean = handle_missing_values(sample_data)

        assert df_clean.isnull().sum().sum() == 0
        assert len(df_clean) == len(sample_data)

    def test_totalcharges_conversion(self):
        """Test TotalCharges conversion."""
        df = pd.DataFrame({
            'TotalCharges': [' ', '100', '200']
        })

        df_clean = handle_missing_values(df)
```

```
        assert df_clean['TotalCharges'].dtype in [np.float64, np.float32]
        assert df_clean['TotalCharges'].iloc[0] == 0.0


class TestDropUnnecessaryColumns:
    """Tests for drop_unnecessary_columns function."""

    def test_drop_single_column(self, sample_data):
        """Test dropping single column."""
        df_clean = drop_unnecessary_columns(sample_data, columns=
['customerID'])

        assert 'customerID' not in df_clean.columns
        assert len(df_clean.columns) == len(sample_data.columns) - 1

    def test_drop_nonexistent_column(self, sample_data):
        """Test dropping column that doesn't exist."""
        df_clean = drop_unnecessary_columns(sample_data, columns=
['fake_col'])

        # Should not raise error
        assert len(df_clean.columns) == len(sample_data.columns)


class TestEncodeTarget:
    """Tests for encode_target function."""

    def test_encode_target_success(self, sample_data):
        """Test target encoding."""
        df_encoded, encoder = encode_target(sample_data,
target_col='Churn')

        assert encoder is not None
        assert df_encoded['Churn'].dtype in [np.int64, np.int32]
        assert set(df_encoded['Churn'].unique()).issubset({0, 1})


class TestSplitFeaturesTarget:
    """Tests for split_features_target function."""

    def test_split_success(self, sample_data):
        """Test splitting features and target."""
        X, y = split_features_target(sample_data, target_col='Churn')

        assert isinstance(X, pd.DataFrame)
        assert isinstance(y, pd.Series)
        assert 'Churn' not in X.columns
        assert len(X) == len(y)

    def test_split_missing_target(self, sample_data):
        """Test error when target is missing."""
        with pytest.raises(ValueError, match="not found"):
            split_features_target(sample_data, target_col='nonexistent')
```

```
@pytest.mark.integration
class TestPreprocessingPipeline:
    """Integration test for complete preprocessing."""

    def test_full_pipeline(self, tmp_path):
        """Test complete preprocessing pipeline."""
        # Create test CSV
        csv_file = tmp_path / "test.csv"
        data = pd.DataFrame({
            'customerID': ['C1', 'C2', 'C3', 'C4', 'C5'],
            'gender': ['Male', 'Female', 'Male', 'Female', 'Male'],
            'tenure': [12, 24, 36, 6, 48],
            'MonthlyCharges': [50, 70, 25, 45, 95],
            'Churn': ['No', 'No', 'Yes', 'No', 'Yes']
        })
        data.to_csv(csv_file, index=False)

        # Run pipeline
        result = preprocess_pipeline(str(csv_file), scale=True,
use_sklearn_pipeline=True)
        X_train, X_test, y_train, y_test, pipeline, encoder = result

        # Verify results
        assert X_train.shape[0] > 0
        assert X_test.shape[0] > 0
        assert pipeline is not None
        assert encoder is not None
```

Run Preprocessing Tests

```
pytest tests/test_preprocess.py -v
```

Part 9.7: Testing Training Module

Create `tests/test_train.py`:

```
"""
Tests for src/train.py module.
"""

import pytest
from sklearn.ensemble import RandomForestClassifier

from src.train import train_random_forest, save_model, evaluate_model
```

```
class TestTrainRandomForest:
    """Tests for train_random_forest function."""

    def test_train_with_default_params(self, processed_data):
        """Test training with default parameters."""
        X_train = processed_data['X_train']
        y_train = processed_data['y_train']

        model = train_random_forest(X_train, y_train,
tune_hyperparameters=False)

        assert isinstance(model, RandomForestClassifier)
        assert hasattr(model, 'estimators_')
        assert model.random_state == 42

    def test_train_can_predict(self, processed_data):
        """Test that trained model can make predictions."""
        X_train = processed_data['X_train']
        y_train = processed_data['y_train']
        X_test = processed_data['X_test']

        model = train_random_forest(X_train, y_train,
tune_hyperparameters=False)
        predictions = model.predict(X_test)

        assert len(predictions) == len(X_test)


class TestSaveModel:
    """Tests for save_model function."""

    def test_save_model_creates_file(self, trained_model, tmp_path):
        """Test that save_model creates a file."""
        model_path = save_model(trained_model, 'test_model',
str(tmp_path))

        assert Path(model_path).exists()
        assert '.pkl' in model_path

    def test_saved_model_can_be_loaded(self, trained_model, tmp_path):
        """Test that saved model can be loaded."""
        import joblib

        model_path = save_model(trained_model, 'test_model',
str(tmp_path))
        loaded_model = joblib.load(model_path)

        assert hasattr(loaded_model, 'predict')


class TestEvaluateModel:
    """Tests for evaluate_model function."""

    def test_evaluate_returns_metrics(self, trained_model,
```

```
processed_data):
        """Test that evaluate returns all metrics."""
        X_test = processed_data['X_test']
        y_test = processed_data['y_test']

        metrics = evaluate_model(trained_model, X_test, y_test)

        assert 'accuracy' in metrics
        assert 'precision' in metrics
        assert 'recall' in metrics
        assert 'f1_score' in metrics
        assert 0.0 <= metrics['accuracy'] <= 1.0


@pytest.mark.parametrize("model_type", [
    'logistic_regression',
    'random_forest',
    'decision_tree'
])
def test_train_different_models(model_type, processed_data):
    """Parametrized test for different model types."""
    from src.train import (
        train_logistic_regression,
        train_random_forest,
        train_decision_tree
    )

    X_train = processed_data['X_train']
    y_train = processed_data['y_train']

    train_functions = {
        'logistic_regression': train_logistic_regression,
        'random_forest': train_random_forest,
        'decision_tree': train_decision_tree
    }

    train_func = train_functions[model_type]
    model = train_func(X_train, y_train, tune_hyperparameters=False)

    assert hasattr(model, 'predict')
```

Part 9.8: Testing Prediction Module

Create `tests/test_predict.py`:

```
"""
Tests for src/predict.py module.
"""

import numpy as np
```

```python
import pytest

from src.predict import load_model, predict, ModelPredictor


class TestLoadModel:
    """Tests for load_model function."""

    def test_load_model_success(self, saved_model_file):
        """Test successful model loading."""
        model = load_model(str(saved_model_file))

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

    def test_load_model_file_not_found(self):
        """Test error for missing model file."""
        with pytest.raises(FileNotFoundError):
            load_model("nonexistent_model.pkl")


class TestPredict:
    """Tests for predict function."""

    def test_predict_returns_array(self, trained_model, processed_data):
        """Test that predict returns numpy array."""
        X_test = processed_data['X_test']

        predictions = predict(trained_model, X_test)

        assert isinstance(predictions, np.ndarray)
        assert len(predictions) == len(X_test)

    def test_predict_empty_input_raises_error(self, trained_model):
        """Test that empty input raises ValueError."""
        X_empty = np.array([])

        with pytest.raises(ValueError, match="empty"):
            predict(trained_model, X_empty)


class TestModelPredictor:
    """Tests for ModelPredictor class."""

    def test_init(self, saved_model_file):
        """Test ModelPredictor initialization."""
        predictor = ModelPredictor(str(saved_model_file))

        assert predictor.model is not None

    def test_predict(self, saved_model_file, processed_data):
        """Test prediction using ModelPredictor."""
        predictor = ModelPredictor(str(saved_model_file))
        X_test = processed_data['X_test']
```

```
        predictions = predictor.predict(X_test)

        assert len(predictions) == len(X_test)
```

Part 9.9: Testing Evaluation Module

Create `tests/test_evaluate.py`:

```
"""
Tests for src/evaluate.py module.
Note: Uses 'y' for true labels and 'yhat' for predictions.
"""

import numpy as np
import pytest

from src.evaluate import calculate_accuracy, calculate_metrics,
evaluate_model


class TestCalculateAccuracy:
    """Tests for calculate_accuracy function."""

    def test_perfect_accuracy(self):
        """Test with perfect predictions."""
        y = np.array(['Yes', 'No', 'Yes', 'No'])
        yhat = np.array(['Yes', 'No', 'Yes', 'No'])

        accuracy = calculate_accuracy(y, yhat)

        assert accuracy == 1.0

    def test_half_accuracy(self):
        """Test with 50% accuracy."""
        y = np.array(['Yes', 'No', 'Yes', 'No'])
        yhat = np.array(['Yes', 'No', 'No', 'Yes'])

        accuracy = calculate_accuracy(y, yhat)

        assert accuracy == 0.5


class TestCalculateMetrics:
    """Tests for calculate_metrics function."""

    def test_all_metrics_present(self):
        """Test that all metrics are calculated."""
        y = np.array(['Yes', 'No', 'Yes', 'No'] * 10)
        yhat = np.array(['Yes', 'No', 'Yes', 'Yes'] * 10)
```

```
        metrics = calculate_metrics(y, yhat)

        assert 'accuracy' in metrics
        assert 'precision' in metrics
        assert 'recall' in metrics
        assert 'f1_score' in metrics
        assert all(0.0 <= v <= 1.0 for v in metrics.values())


class TestEvaluateModel:
    """Tests for evaluate_model function."""

    def test_complete_evaluation(self, trained_model, processed_data):
        """Test complete model evaluation."""
        X_test = processed_data['X_test']
        y_test = processed_data['y_test']

        results = evaluate_model(trained_model, X_test, y_test)

        assert 'accuracy' in results
        assert 'confusion_matrix' in results
        assert 'classification_report' in results
```

Part 9.10: Integration Tests

Create `tests/test_integration.py`:

```
"""
Integration tests for complete ML workflows.
"""

import pytest
from pathlib import Path

from src.preprocess import preprocess_pipeline
from src.train import train_random_forest, save_model
from src.predict import ModelPredictor, predict
from src.evaluate import evaluate_model


@pytest.mark.integration
class TestCompleteMLPipeline:
    """Tests for complete ML pipeline."""

    def test_csv_to_predictions(self, sample_csv_file_large, tmp_path):
        """Test complete workflow: CSV → preprocessing → training →
prediction."""
        # Preprocess
        result = preprocess_pipeline(
```

```
            str(sample_csv_file_large),
            scale=True,
            use_sklearn_pipeline=True
        )
        X_train, X_test, y_train, y_test, pipeline, encoder = result

        # Train
        model = train_random_forest(X_train, y_train,
tune_hyperparameters=False)

        # Predict
        predictions = predict(model, X_test)

        # Evaluate
        results = evaluate_model(model, X_test, y_test)

        # Verify
        assert len(predictions) == len(X_test)
        assert 0.0 <= results['accuracy'] <= 1.0

    def test_save_load_workflow(self, sample_csv_file_large, tmp_path):
        """Test saving and loading all components."""
        # Preprocess
        X_train, X_test, y_train, y_test, pipeline, encoder =
preprocess_pipeline(
            str(sample_csv_file_large),
            scale=True,
            use_sklearn_pipeline=True
        )

        # Train and save
        model = train_random_forest(X_train, y_train,
tune_hyperparameters=False)
        model_path = save_model(model, 'test_model', str(tmp_path))

        # Load and predict
        predictor = ModelPredictor(model_path)
        predictions = predictor.predict(X_test)

        assert len(predictions) == len(X_test)
```

Part 9.11: Running Tests and Code Coverage

Run All Tests

```
# Run all tests
pytest

# Run with verbose output
pytest -v
```

```
# Run specific test file
pytest tests/test_preprocess.py

# Run specific test
pytest tests/test_preprocess.py::test_load_data_success

# Run tests by marker
pytest -m unit
pytest -m integration
```

Measure Code Coverage

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

# Open coverage report in browser
open htmlcov/index.html  # Mac
xdg-open htmlcov/index.html  # Linux
start htmlcov/index.html  # Windows
```

Expected output:

```
----------- coverage: platform linux, python 3.12.12-final-0 -----------
Name                        Stmts   Miss  Cover
-----------------------------------------------
src/__init__.py                 1      0   100%
src/evaluate.py               150     15    90%
src/predict.py                120     10    92%
src/preprocess.py             200     20    90%
src/train.py                  250     25    90%
src/utils/config.py            10      0   100%
-----------------------------------------------
TOTAL                         731     70    90%
```

Coverage targets:

- **80%+**: Acceptable
- **90%+**: Good
- **95%+**: Excellent
- **100%**: Overkill (don't aim for this)

Understanding Coverage Report

The HTML report shows:

- **Green lines**: Covered by tests

- **Red lines**: Not covered
- **Yellow lines**: Partially covered

Focus on covering:

- ✅ Critical business logic
- ✅ Error handling paths
- ✅ Edge cases

Don't worry about:

- ❌ Simple getters/setters
- ❌ `__init__` methods with just assignments
- ❌ Logging statements

Part 9.12: Best Practices for ML Testing

1. Use Small, Fast Test Data

```
# ❌ Bad: Using full dataset
@pytest.fixture
def test_data():
    return pd.read_csv('data/raw/full_dataset.csv')  # 1GB file!

# ✅ Good: Using small synthetic data
@pytest.fixture
def test_data():
    return pd.DataFrame({
        'feature1': [1, 2, 3, 4, 5],
        'feature2': ['a', 'b', 'c', 'd', 'e'],
        'target': [0, 1, 0, 1, 0]
    })
```

2. Mock External Dependencies

```
# Mock MLflow to test without actually logging
@patch('mlflow.log_param')
@patch('mlflow.log_metric')
def test_train_with_mlflow(mock_metric, mock_param):
    model = train_model(X, y)

    assert mock_param.called
    assert mock_metric.called
```

3. Test Edge Cases

```
def test_empty_dataframe():
    """Test with empty input."""
    df = pd.DataFrame()

    with pytest.raises(ValueError, match="empty"):
        preprocess(df)

def test_single_sample():
    """Test with single sample."""
    X = np.array([[1, 2, 3]])
    predictions = model.predict(X)

    assert len(predictions) == 1
```

4. Use Parametrized Tests

```
@pytest.mark.parametrize("input_val,expected", [
    (0, 0),
    (1, 1),
    (5, 25),
    (-2, 4)
])
def test_square(input_val, expected):
    assert square(input_val) == expected
```

5. Organize Tests into Classes

```
class TestDataLoading:
    """All tests for data loading."""

    def test_load_csv(self):
        pass

    def test_load_excel(self):
        pass

    def test_load_json(self):
        pass
```

6. Don't Test Implementation Details

```
# ❌ Bad: Testing internal variables
def test_internal_state():
    model = Model()
    assert model._internal_counter == 0  # Don't test this
```

```python
# ✅ Good: Testing behavior
def test_prediction_output():
    model = Model()
    result = model.predict(X)
    assert len(result) == len(X)  # Test this
```

Part 9.13: Continuous Integration (Preview)

What is CI?

Continuous Integration (CI) automatically runs your tests every time you push code to GitHub.

Benefits:

- ✅ Catch bugs before merging
- ✅ Ensure all contributors run tests
- ✅ Maintain code quality
- ✅ Prevent broken code in main branch

Preview: GitHub Actions

In Lab 06, you'll set up CI with GitHub Actions. Here's a preview:

`.github/workflows/tests.yml`:

```yaml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.12'

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

    - name: Run tests
      run: |
        pytest --cov=src --cov-report=term
```

This will:

- Run tests on every push
- Show test results in GitHub
- Block merges if tests fail

Part 9.14: Troubleshooting Common Issues

Issue: Tests not discovered

Problem: `pytest` finds no tests

Solution:

```
# Check test discovery
pytest --collect-only

# Ensure files start with 'test_'
mv my_tests.py test_my_module.py

# Ensure functions start with 'test_'
def test_my_function():  # Good
def my_test():  # Bad - won't be discovered
```

Issue: Import errors

Problem: `ModuleNotFoundError: No module named 'src'`

Solution:

```
# Run pytest from project root
cd /path/to/project
pytest

# Or install your package in editable mode
pip install -e .
```

Issue: Fixture not found

Problem: `fixture 'sample_data' not found`

Solution:

- Ensure fixture is in `conftest.py`
- Check fixture name matches parameter name
- Verify `conftest.py` is in tests directory

Issue: Tests pass locally but fail in CI

Problem: Different behavior in CI

Common causes:

- Different Python version
- Different package versions
- Different random seeds
- Timezone differences

Solution: Use same Python version and pin package versions

Part 9.15: Summary and Next Steps

What You've Accomplished

Congratulations! You've now completed Lab 02 in its entirety. Your project now has:

✅ **Virtual environments** for reproducibility ✅ **CLI interfaces** for easy usage ✅ **Hyperparameter tuning** for better models ✅ **YAML configuration** for flexibility ✅ **DVC** for data version control ✅ **MLflow** for experiment tracking ✅ **Comprehensive test suite** with pytest

Your test suite includes:

- ~2,700 lines of test code
- Unit tests for all modules
- Integration tests for complete workflows
- Test fixtures for reusable data
- Code coverage measurement
- Organized test structure

Quality Metrics

Your project should now achieve:

- **Code coverage**: >80%
- **Tests**: 50+ test functions
- **Test files**: 6 comprehensive test modules
- **Documentation**: Tests serve as examples

Commands Reference

```
# Run all tests
pytest

# Run with coverage
pytest --cov=src --cov-report=html --cov-report=term
```

```
# Run specific tests
pytest tests/test_preprocess.py
pytest tests/test_train.py -v
pytest -m integration

# Run tests matching pattern
pytest -k "test_load"

# Show slowest tests
pytest --durations=10
```

Project Status: Lab 02 Complete! 🎉

Version: 2.3.0 (with Testing)

Next Labs:

- **Lab 03**: REST API Development (Flask/FastAPI)
- **Lab 04**: Containerization (Docker)
- **Lab 05**: Cloud Deployment (AWS/GCP/Azure)
- **Lab 06**: CI/CD & Monitoring (GitHub Actions)

Commit Your Work

```
git add tests/ pytest.ini
git commit -m "feat: Add comprehensive pytest test suite (Lab 02 Part 9)

**Testing Infrastructure**:
- Add pytest and pytest-cov to requirements.txt
- Create tests/ directory with fixtures
- Configure pytest with pytest.ini

**Test Coverage**:
- test_preprocess.py: Data loading, pipelines, encoding
- test_train.py: Model training, MLflow integration
- test_predict.py: Predictions, file I/O
- test_evaluate.py: Metrics calculation
- test_integration.py: End-to-end workflows

**Documentation**:
- Add Lab 02 Part 9 testing tutorial
- Update README with testing section

**Results**:
- Coverage: >80%
- All tests pass
- Ready for CI/CD (Lab 06)

Lab 02 Complete: Structure + DVC + MLflow + Testing
Version: 2.3.0"
```

```
git push
```

Additional Resources

Testing Books

- "Test-Driven Development with Python" by Harry Percival
- "Python Testing with pytest" by Brian Okken

Online Resources

- pytest Documentation: https://docs.pytest.org/
- Real Python pytest Tutorial: https://realpython.com/pytest-python-testing/
- Effective Python Testing: https://effectivepython.com/

Testing Tools

- **pytest**: Main testing framework
- **pytest-cov**: Coverage measurement
- **pytest-mock**: Mocking utilities
- **hypothesis**: Property-based testing
- **tox**: Testing across multiple environments

Part 10: Submission

What to Submit

For this lab assignment, you must **submit a single zip file** containing the required screenshots to the course Moodle page. Additionally, ensure you complete the Git commit and DVC access requirements directly in their respective platforms (GitHub and DagsHub/Google Drive).

1. **Git Commit**:
    - Make one or more **meaningful commits** to your GitHub Classroom repository that include all the code changes implementing the requirements of this lab (CLI interfaces, DVC integration, MLflow integration, updated `requirements.txt`, etc., tests are optional). Your instructor will review your repository directly.
2. **DVC Remote Storage Access**:
    - **If using DagsHub (Recommended)**: Add your instructor as a collaborator to your DagsHub repository. Your instructor's DagsHub username is `ajallooe`.
    - **If using Google Drive**: Share the Google Drive folder you configured as your DVC remote with your instructor (provide their email address if needed, or follow course instructions for sharing).
    - Ensure your data has been successfully pushed using `dvc push`.

3. **Screenshots (Include in Zip File)**:
    - **MLflow UI - Multiple Runs**: A screenshot showing the MLflow UI with your experiment selected, displaying the table view with at least **2-3 distinct runs** resulting from different parameters or configurations.
    - **MLflow UI - Single Run Details**: A screenshot showing the detail page for **one specific run**. This screenshot must clearly display the logged **Parameters**, **Metrics**, and **Artifacts** (showing at least the logged model folder, e.g., "model").
    - **DVC Push Confirmation**: A screenshot of your terminal after successfully running `dvc push`. The output should clearly indicate that your data files were pushed or are already up-to-date with the remote.
        - *How to get this:* After making sure your DVC remote is configured and you have tracked your data (`dvc add data/raw data/processed`), run `dvc push` in your terminal. Capture the output that typically looks like `Pushing...`, `X files pushed`, or confirms files are `Alreadyincache/Up-to-date`.

Submission Summary:

- **On Moodle**: Submit **one zip file** containing the three required screenshots (MLflow multiple runs, MLflow single run details, DVC push confirmation).
- **On GitHub**: Ensure your latest code, including DVC (`.dvc`, `.gitignore`) and MLflow integration, is **committed and pushed** to your GitHub Classroom repository.
- **On DagsHub/Google Drive**: Ensure your DVC remote is **shared with/accessible to** the instructor (`ajallooe` for DagsHub).

Note: Grading is based on the successful implementation of CLI, DVC, MLflow, and testing as reflected in your GitHub repository, the confirmation of DVC remote setup and push, and the evidence provided in the MLflow screenshots.

Congratulations!🎉

You've completed the second major step in your ML deployment journey! You've taken your structured project and elevated it with professional-grade tools:

✅ **Data Version Control (DVC)** for reproducible data management
✅ **Experiment Tracking (MLflow)** for tracking, comparing, and managing models
✅ **Automated Testing (pytest)** for reliable, maintainable code
✅ **Functional CLIs** to make your scripts usable and configurable

Key Takeaways:

1. ✅ **Git is for code, DVC is for data.** Never commit large data files to Git.
2. ✅ **If you can't reproduce it, it's not finished.** DVC ensures your data is versioned just like your code.
3. ✅ **Track *every* experiment.** MLflow is your lab notebook, preventing "magic" models you can't recreate.
4. ✅ **If it's not tested, it's broken.** Tests are your safety net, allowing you to refactor and add features without fear.

5. ✅ **Scripts should be configurable.** CLIs (`argparse`) separate configuration from code logic.

You're now ready to:

- **Lab 3**: Serve your model as a **REST API** with Flask/FastAPI
- **Lab 4**: Package your API into a **Docker container**
- **Lab 5**: **Deploy** your container to the cloud
- **Lab 6**: Implement **CI/CD** and **Monitoring**

The foundation you've built and strengthened in this lab is what separates a data science experiment from a deployable machine learning product.

Lab 2 Complete! 🎊

Your code is production-ready, maintainable, and reliable. Well done!

Lab 2 Instructions
CMPT 2500: Machine Learning Deployment and Software Development
NorQuest College