Creating a Reproducible and Maintainable Machine Learning Environment

Hung-Hsuan Chen

Department of Computer Science and Information Engineering
National Central University
hhchen1105@acm.org

Abstract

Reproducibility and maintainability are critical challenges in machine learning research and development. This tutorial provides a guide to setting up a repeatable experimental environment using modern tools and techniques. Specifically, it covers version control using Git, experiment tracking with MLflow, creating isolated experiment environments with Conda for virtual environments and Docker for containerization, testing with pytest, and packaging machine learning code into a reusable library. By the end of this tutorial, practitioners and researchers will have the foundational knowledge to ensure that their machine learning workflows are reproducible and maintainable.

Contents

1 Introduction **4**

2 Version Control with Git **4**
- 2.1 Basic Git Commands . . . 5
- 2.2 Using GitHub . . . 5
- 2.3 Fixing Merge Conflicts in Git . . . 5
  - 2.3.1 Identifying the Conflict . . . 5
  - 2.3.2 Viewing the Conflict . . . 5
  - 2.3.3 Resolving the Conflict . . . 6
  - 2.3.4 Aborting the Merge (Optional) . . . 6
  - 2.3.5 Verifying the Merge . . . 6
  - 2.3.6 Best Practices for Avoiding Conflicts . . . 6

3 Experiment Tracking with MLflow **7**
- 3.1 Tracking an Experiment . . . 7

4 Experiment Environment with Conda **8**
- 4.1 Creating a Conda Environment . . . 8
  - 4.1.1 Installing Conda . . . 8
  - 4.1.2 Creating and Managing Environments . . . 8
  - 4.1.3 Installing Packages . . . 9
  - 4.1.4 Managing Environments and Dependencies . . . 9
  - 4.1.5 Using Pip Inside Conda Environments . . . 9
- 4.2 Best Practices for Conda Environments . . . 10

5 Experiment Environment with Docker **10**
- 5.1 Setting Up Docker . . . 10
- 5.2 Creating a Dockerfile . . . 10
- 5.3 Building and Running the Docker Container . . . 11
- 5.4 Sharing Docker Images . . . 11
- 5.5 Best Practices for Docker . . . 11

6 Testing and Automation with pytest **12**
- 6.1 Basic pytest Commands . . . 12
- 6.2 Writing Tests . . . 12
- 6.3 How pytest Discovers Test Cases . . . 12
  - 6.3.1 Test Discovery Rules . . . 12
  - 6.3.2 Customizing Test Discovery . . . 13
  - 6.3.3 Ignoring Files or Directories . . . 13
  - 6.3.4 Checking Discovery without Running Tests . . . 14

7 Packaging Code into a Library **14**
- 7.1 Setting Up the Package Structure . . . 14
- 7.2 Writing `setup.py` . . . 14
- 7.3 Building and Installing the Package . . . 14

8 Conclusion **15**

1 Introduction

Machine learning research and development often involve iterative experimentation, in which code, data, and environments change frequently. Without proper tools and practices, ensuring the reproducibility and maintainability of these experiments becomes increasingly difficult. Inconsistent environments, poorly tracked code changes, and lack of automated testing are some of the common issues researchers and engineers face.

This tutorial introduces essential tools and methodologies that address these challenges. The tutorial is aimed at machine learning practitioners and researchers who wish to establish a robust, repeatable workflow for their experiments. The focus is on six key areas:

- **Version Control with Git:** Git is a powerful distributed version control system that tracks code changes and enables collaboration between individuals.
- **Experiment Tracking with MLflow:** MLflow allows for efficient tracking of experiments, logging metrics and parameters, ensuring reproducibility across multiple runs.
- **Environment Management with Conda:** Conda provides a flexible solution for managing virtual environments and package dependencies, ensuring that the same setup can be replicated across different systems.
- **Containerization with Docker:** Docker allows complete isolation of applications and their dependencies, providing a consistent environment to run machine learning experiments on different platforms.
- **Testing with pytest:** Writing automated tests ensures that the code functions correctly and consistently as changes are introduced, making pytest an essential tool for robust machine learning pipelines.
- **Packaging Code into a Library:** Packaging machine learning code into reusable libraries makes it easier to distribute and reuse across different projects and teams.

The following sections will provide step-by-step instructions on using these tools and examples illustrating their practical use in machine learning workflows. By adopting the techniques outlined in this tutorial, you will be equipped to develop machine learning projects that are reproducible and easier to scale and maintain in the long run.

2 Version Control with Git

Version control helps track and manage changes in your code. Git is a distributed version control system that enables collaboration and version management.

2.1 Basic Git Commands

```
# Initialize a Git repository
git init

# Add files to the staging area
git add <file_name>

# Commit changes with a message
git commit -m "Initial commit"

# Check the status of the repository
git status

# Push changes to a remote repository
git push origin main
```

2.2 Using GitHub

GitHub is a popular platform for hosting Git repositories. After setting up your repository on GitHub, you can clone it locally:

```
git clone https://github.com/username/repository.git
```

2.3 Fixing Merge Conflicts in Git

Merge conflicts occur when changes made to the same parts of a file in different branches or by other contributors cannot be automatically merged by Git. Here is how you can resolve a merge conflict.

2.3.1 Identifying the Conflict

When you attempt to merge two branches that have conflicting changes, Git will alert you with a message like:

```
Auto-merging <file_name>
CONFLICT (content): Merge conflict in <file_name>
Automatic merge failed; fix conflicts and then commit the result
    .
```

At this point, Git has stopped the merge and marked the file as conflicted.

2.3.2 Viewing the Conflict

To see where the conflict occurred, open the conflicting file in a text editor. You will see conflict markers like this:

```
<<<<<<< HEAD
# Changes from your current branch (HEAD)
print("Hello from branch A")
=======
```

```
# Changes from the branch you're merging into your branch
print("Hello from branch B")
>>>>>>> <branch_name>
```

The section between '<<<<<<< HEAD' and '=======' represents the changes in your current branch, while the section between '=======' and '>>>>>>> <branch_name>' represents the changes from the branch you are trying to merge.

2.3.3 Resolving the Conflict

Manually edit the file to resolve the conflict. For example, you might combine the two versions or choose one over the other:

```
# Merged content after resolving the conflict
print("Hello from both branches")
```

After you've made your changes and resolved the conflict, you need to stage and commit the resolved file:

```
# Stage the resolved file
git add <file_name>

# Commit the resolution
git commit -m "Resolved␣merge␣conflict␣in␣<file_name>"
```

2.3.4 Aborting the Merge (Optional)

If you decide that you do not want to continue with the merge, you can abort it and revert to the state before the merge was attempted:

```
# Abort the merge process
git merge --abort
```

This command will stop the merge and restore your working directory to its previous state.

2.3.5 Verifying the Merge

After resolving the conflict and committing the changes, you can check the status of your repository:

```
git status
```

If all conflicts have been resolved, Git will report that your branch is clean, and you can continue working as usual.

2.3.6 Best Practices for Avoiding Conflicts

To minimize the likelihood of conflicts:

- Frequently pull the latest changes from the remote repository before making your own changes.
- Commit and push your changes regularly to ensure that others can access them.
- When collaborating with a team, communicate with team members about which parts of the codebase you are working on to avoid overlapping changes.

3 Experiment Tracking with MLflow

MLflow helps you manage the entire lifecycle of machine learning models, including experimentation, reproducibility, and deployment.

3.1 Tracking an Experiment

Install MLflow with the following commands:

```
pip install mlflow
```

Start the MLflow server at localhost by

```
mlflow server --host 127.0.0.1 --port 8080
```

You can log metrics, parameters, and models with MLflow in your code:

```
import mlflow

# Train a logistic regression classifier
params = {
  "solver": "lbfgs",
  "max_iter": 1000,
  "multi_class": "auto",
  "random_state": 8888,
}
lr = LogisticRegression(**params)
y_pred = lr.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Start logging
mlflow.set_tracking_uri(uri="http://127.0.0.1:8080")
mlflow.set_experiment("First␣experiment")
with mlflow.start_run():
    mlflow.log_param(params)
    mlflow.log_metric({
        "accuracy": accuracy,
    })
    mlflow.log_model(lr, "model.pkl")
```

4 Experiment Environment with Conda

Managing environments ensures consistency across machines and operating systems. Conda is a powerful tool for managing virtual environments and package dependencies, ensuring that your machine learning projects are reproducible and portable.

4.1 Creating a Conda Environment

Conda is a package manager and environment management system that works on Windows, macOS, and Linux. It enables you to create isolated environments for your projects, ensuring that dependencies do not conflict with each other across different projects.

4.1.1 Installing Conda

First, install Conda if you haven't already. Conda is bundled with the Anaconda or Miniconda distributions. While Anaconda includes many packages by default, Miniconda is a smaller alternative that allows you to install only the necessary packages.

- Download Miniconda from `https://docs.conda.io/en/latest/miniconda.html` and follow the installation instructions.
- Once installed, verify the installation with the command:

```
conda --version
```

4.1.2 Creating and Managing Environments

List the available Python versions by the following command:

```
conda search python
```

To create a new environment for your machine learning project, use the following command:

```
# Create a new conda environment with Python 3.8
conda create --name ml_env python=3.8
```

This command will create a new environment named `ml_env` with Python version 3.8. You can specify any Python version according to the requirements of your project.

List the available environments by:

```
conda env list
```

To activate the environment, run:

```
conda activate ml_env
```

This switches your shell to the newly created environment. Any packages that you now install will be isolated within this environment.

To deactivate and return to the base environment:

```
conda deactivate
```

4.1.3 Installing Packages

Once inside the environment, you can install the required packages for your project. Conda can install many commonly used machine learning libraries directly:

```
# Install popular machine learning packages
conda install numpy pandas scikit-learn matplotlib
```

If the package you need is not available in the default Conda channels, you can use the Conda Forge channel, which includes a broader range of community-supported packages:

```
# Installing a package from conda-forge
conda install -c conda-forge tensorflow
```

4.1.4 Managing Environments and Dependencies

You can list all environments on your system with:

```
conda info --envs
```

To remove an environment that you no longer need:

```
conda env remove --name ml_env
```

Conda also provides a way to export the environment's configuration, which is useful for sharing with collaborators or deploying to a new machine. Use the following command to generate an `environment.yml` file:

```
# Export environment configuration to a YAML file
conda env export > environment.yml
```

This file can be used to recreate the environment on another system:

```
# Create an environment from a YAML file
conda env create -f environment.yml
```

4.1.5 Using Pip Inside Conda Environments

While Conda manages most packages, some Python packages are only available through `pip`. You can still install them within a Conda environment by simply running:

```
pip install <package_name>
```

However, it is a best practice to first try installing with Conda before resorting to `pip` to avoid potential conflicts.

4.2 Best Practices for Conda Environments

Here are a few best practices for managing your Conda environments effectively:

- Always specify exact versions of critical dependencies in your `environment.yml` file to ensure reproducibility.
- Regularly update your environment using:

```
conda update --all
```

- Keep environments project-specific to avoid clutter and dependency conflicts.
- Use environment variables to manage environment-specific settings such as data paths or API keys.

5 Experiment Environment with Docker

Docker is a platform for developing, shipping, and running applications in containers. Containers allow you to bundle your application and its dependencies into a single portable unit. This ensures that your machine learning models and environments can run consistently on any platform.

5.1 Setting Up Docker

To get started with Docker, follow these steps to install and verify your setup:

- Download Docker from `https://www.docker.com/`.
- Follow the installation instructions for your operating system.
- Verify the installation by running:

```
docker --version
```

5.2 Creating a Dockerfile

A `Dockerfile` is a text document that contains all the commands to assemble a Docker image. Below is an example `Dockerfile` for a machine learning project:

```
# Start with the official Python image
FROM python:3.8-slim

# Set the working directory in the container
WORKDIR /app

# Copy the current directory contents into the container
COPY . /app

# Install any required dependencies from the requirements file
```

```
RUN pip install --no-cache-dir -r requirements.txt

# Make port 5000 available to the world outside this container
EXPOSE 5000

# Define the command to run the application
CMD ["python", "app.py"]
```

This `Dockerfile` sets up a Python 3.8 environment, installs the required dependencies from `requirements.txt`, and defines a simple command to run the application.

5.3 Building and Running the Docker Container

Once you have written the `Dockerfile`, you can build a Docker image from it. This image can be shared or deployed anywhere Docker is installed.

```
# Build the Docker image
docker build -t my-ml-app .

# Run the Docker container
docker run -p 8080:80 my-ml-app
```

This command will build your Docker image, naming it `my-ml-app`, and run it while mapping port 8080 inside the container to port 80 on your local machine.

5.4 Sharing Docker Images

Once your Docker image is built, you can push it to the Docker Hub or any private registry, allowing others to pull and run the same environment.

```
# Login to Docker Hub
docker login

# Tag the image
docker tag my-ml-app username/my-ml-app

# Push the image to Docker Hub
docker push username/my-ml-app
```

5.5 Best Practices for Docker

Here are a few best practices when working with Docker:

- Keep your `Dockerfile` clean by using small, optimized base images such as `python:3.8-slim`.
- Use multi-stage builds to reduce image size by separating the build and runtime stages.

- Regularly update your images to include the latest security patches and dependencies.
- Avoid hard-coding configuration variables inside your container. Use environment variables instead.

6 Testing and Automation with pytest

Testing is an essential part of software development. `pytest` is a popular testing framework for Python.

6.1 Basic pytest Commands

```
# Install pytest
pip install pytest

# Run all tests in a directory
pytest

# Run a specific test file
pytest test_file.py
```

6.2 Writing Tests

Create test cases in a separate file (e.g., `test_model.py`):

```
def test_example():
    assert 1 + 1 == 2
```

6.3 How pytest Discovers Test Cases

One of the powerful features of `pytest` is its ability to automatically discover and test cases without additional configuration. The `pytest` follows certain conventions to identify and collect tests from the codebase.

6.3.1 Test Discovery Rules

By default, `pytest` follows these rules to discover test cases:

- **Test File Naming**: The files containing test cases should be named using the prefix `test_` or the suffix `_test`. For example:

```
    test_example.py
    example_test.py
```

- **Test Function Naming**: The functions that represent test cases should also be named with the prefix `test_`. For example:

```
    def test_addition():
        assert 1 + 1 == 2
```

- **Test Classes**: The test classes should be named with a prefix of `Test` (though this is not strictly required). The class should not have an `__init__` method:

```
    class TestMathOperations:
        def test_multiplication(self):
            assert 2 * 3 == 6
```

- **Directories**: By default, pytest recursively searches all subdirectories for test files, as long as the directory name does not begin with `_`.

6.3.2 Customizing Test Discovery

In cases where your test files or functions do not follow the default conventions, you can customize how pytest discovers tests:

- **Running Specific Tests**: You can directly specify the directory, file, or test case you want to run:

```
    # Run all tests in a specific directory
    pytest path/to/tests/

    # Run a specific test file
    pytest test_specific.py

    # Run a specific test function within a file
    pytest test_specific.py::test_function_name
```

- **Changing Discovery Patterns**: You can modify pytest's discovery behavior by using the options in the `pytest.ini` configuration file. For example, you can change the test discovery patterns:

```
    [pytest]
    python_files = *_testcase.py
    python_functions = check_*
```

  This configuration will instruct pytest to look for files ending with `_testcase.py` and functions starting with `check_` instead of the default `test_`.

6.3.3 Ignoring Files or Directories

You can instruct `pytest` to ignore specific directories or files during discovery. To ignore a directory, you can include a `conftest.py` file in the directory and add the following:

```
collect_ignore = ["ignored_directory/", "ignored_file.py"]
```

Alternatively, you can use the `--ignore` flag when running tests:

```
pytest --ignore=path/to/ignored_directory/
```

6.3.4 Checking Discovery without Running Tests

To see which test cases `pytest` has discovered without running them, you can use the `--collect-only` option:

```
pytest --collect-only
```

This will output a list of all test cases that `pytest` has discovered based on the default or customized discovery rules.

7 Packaging Code into a Library

Packaging your code allows you to distribute it as a library.

7.1 Setting Up the Package Structure

```
my_package/
    |-- my_module.py
    |-- __init__.py
    |-- setup.py
    |-- tests/
        |-- test_my_module.py
```

7.2 Writing `setup.py`

```
from setuptools import setup, find_packages

setup(
    name="my_package",
    version="0.1",
    packages=find_packages(),
    install_requires=[
        "numpy",
        "pandas",
    ],
)
```

7.3 Building and Installing the Package

```
# Build the package
python setup.py build

# Install the package
python setup.py install
```

8 Conclusion

By mastering version control with Git, experiment tracking with MLflow, setting up virtual environments with Conda, containerizing applications with Docker, testing with pytest, and packaging your code into reusable libraries, you can create a highly reproducible and maintainable machine learning workflow. Each of these tools and techniques addresses a specific challenge in the development and deployment of machine learning projects, ensuring that your work is not only reliable and scalable, but also easy to share and reproduce across different platforms and collaborators.