Reproducibility & Environments

Week 8 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra

IIT Gandhinagar

The "Works on My Machine" Problem

You built a Netflix movie predictor. It works great on your laptop.

Your friend tries to run it:

```
ImportError: No module named 'sklearn'
```

You say: "Just pip install sklearn"

```
ERROR: Could not find a version that satisfies the requirement sklearn
```

3 hours later: Still debugging Python versions, missing dependencies...

Sound familiar?

Why Reproducibility Matters

For you:

- 6 months later, you can still run your own code
- Switch laptops without days of setup
- Debug issues consistently

For collaboration:

- Teammates can run your code immediately
- No more "but it works for me!"
- Onboard new team members quickly

For science:

- Others can verify your results
- Build on your work

The Reproducibility Spectrum

Your code is only as good as its ability to run elsewhere. If no one else can run it, it might as well not exist. Reproducibility isn't about being fancy - it's about being useful.

```
                        Reproducibility Spectrum

    Not Reproducible                                    Fully Reproducible
         |                                                      |
         v                                                      v
    [Just code] → [+ README] → [+ requirements.txt] → [+ Docker] → [+ CI/CD]
         ↓             ↓                ↓                    ↓            ↓
     "What?"     "Maybe..."      "Probably!"          "Definitely"   "Automated"
```

Today's goal: Get you to "Probably!" or better.

Connection to Our Netflix Project

```
Week 1-7: Built a movie success predictor
          ↓
Week 8:   Make it reproducible!
          - Anyone can run your code
          - Same results every time
          - Works on any machine
```

Goal: Package our Netflix project so anyone can use it.

Part 1: Virtual Environments

Keeping projects separate

The Problem: Dependency Conflicts

Scenario:

Project	Python	TensorFlow	NumPy
Netflix Predictor	3.10	2.12	1.24
Old School Project	3.8	1.15	1.19
Your System	3.11	???	???

Can't install both TensorFlow versions on the same system!

Solution: Give each project its own isolated environment.

Virtual Environments: The Concept

Think of it like separate rooms in a house:

```
Your Computer
├── Project A's Room
│   └── Python 3.10, TensorFlow 2.12, NumPy 1.24
│
├── Project B's Room
│   └── Python 3.8, TensorFlow 1.15, NumPy 1.19
│
└── Living Room (system Python)
    └── Python 3.11 (don't touch this!)
```

Each room has its own stuff. No conflicts!

Creating a Virtual Environment

Step 1: Create the environment

```
python -m venv netflix_env
```

Step 2: Activate it

```
# Mac/Linux
source netflix_env/bin/activate

# Windows
netflix_env\Scripts\activate
```

Step 3: Your prompt changes

```
(netflix_env) $ python --version
Python 3.10.12
```

Now you're in the Netflix room!

Installing Packages in Your Environment

With the environment activated:

```
# Install what you need
pip install pandas scikit-learn matplotlib

# Check what's installed
pip list

# When done, deactivate
deactivate
```

Key insight: Packages only install in the active environment.

Your system Python stays clean!

requirements.txt: Your Shopping List

Save your dependencies:

```
pip freeze > requirements.txt
```

What it creates:

```
numpy==1.24.3
pandas==2.0.2
scikit-learn==1.2.2
matplotlib==3.7.1
```

Anyone can now install exactly what you have:

```
pip install -r requirements.txt
```

Good vs Bad requirements.txt

Good (pinned versions):

```
numpy==1.24.3
pandas==2.0.2
scikit-learn==1.2.2
```

Bad (unpinned):

```
numpy
pandas
scikit-learn
```

Why? Tomorrow, scikit-learn 2.0 releases with breaking changes. Your code breaks for new users, but not for you.

Pin your versions for reproducibility!

Conda: An Alternative

Conda is popular in data science. It can manage:

- Python versions (not just packages)
- Non-Python dependencies (CUDA, C libraries)

```
# Create environment with specific Python
conda create -n netflix python=3.10

# Activate
conda activate netflix

# Install packages
conda install pandas scikit-learn

# Export environment
conda env export > environment.yml

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

venv vs Conda: Which to Use?

Feature	venv	Conda
Built into Python	Yes	No (install separately)
Manage Python versions	No	Yes
Non-Python packages	No	Yes (CUDA, etc.)
Speed	Fast	Slower
File	requirements.txt	environment.yml

Recommendation for this course: Start with venv (simpler).

Use Conda when you need GPU/CUDA setup.

Part 2: Random Seeds

Getting the same results every time

The Randomness Problem

Run your Netflix model training twice:

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

X_train, X_test, y_train, y_test = train_test_split(X, y)
model = RandomForestClassifier()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
```

Run 1: 0.82
Run 2: 0.79
Run 3: 0.84

Which result do you report?

What's Random in ML?

Many operations use random numbers:

1. **Train/test split** - which samples go where?
2. **Model initialization** - starting weights
3. **Shuffling data** - order during training
4. **Dropout** - which neurons to drop
5. **Data augmentation** - random transformations

Without control: Different results every run.

Setting Random Seeds

Simple fix: Tell Python what random numbers to use.

```python
import random
import numpy as np
from sklearn.model_selection import train_test_split

# Set the seed ONCE at the start
random.seed(42)
np.random.seed(42)

# Now this split is reproducible
X_train, X_test, y_train, y_test = train_test_split(
    X, y, random_state=42
)
```

Run it 100 times → Same split every time!

A Complete Seed Function

```python
import random
import numpy as np

def set_seed(seed=42):
    """Set all random seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)

    # If using PyTorch
    try:
        import torch
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    except ImportError:
        pass
```

Why 42? It's a tradition (Hitchhiker's Guide to the Galaxy).

Any number works!

Don't Forget random_state!

Many sklearn functions have a `random_state` parameter:

```
# Train/test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# Random Forest
model = RandomForestClassifier(
    n_estimators=100, random_state=42
)

# Cross-validation with shuffling
cross_val_score(model, X, y, cv=5, random_state=42)  #
❌
 No!

# Use a fixed KFold instead
```

Part 3: Docker Basics

"Works on my machine" → "Works on EVERY machine"

Virtual Environments Aren't Enough

Scenario: You share your requirements.txt, but...

- Friend has different OS (Windows vs Mac vs Linux)
- System libraries differ
- CUDA versions conflict
- Even PATH configurations vary

Virtual environments isolate Python, not the whole system.

Docker: Package Everything

Docker creates a container with:

- Operating system
- Python version
- All libraries
- Your code
- Configuration

It's like shipping your entire laptop to someone!

```
Your Code + Python + Linux + Everything
            ↓
        Container
            ↓
    Runs identically everywhere
```

Docker Concepts

Term	What It Is	Analogy
Image	Blueprint/template	Recipe
Container	Running instance	Cooked dish
Dockerfile	Instructions to build image	Recipe card
Registry	Store for images	Recipe book

Workflow:

```
Dockerfile → (build) → Image → (run) → Container
```

Your First Dockerfile

Create a file named `Dockerfile` (no extension):

```
# Start from a Python image
FROM python:3.10-slim

# Set working directory
WORKDIR /app

# Copy requirements first (for caching)
COPY requirements.txt .

# Install dependencies
RUN pip install -r requirements.txt

# Copy your code
COPY . .

# Command to run
```

Building and Running

Build the image:

```
docker build -t netflix-predictor .
```

Run it:

```
docker run netflix-predictor
```

That's it! Your code runs in an isolated container.

Works on any machine with Docker installed.

Common Docker Commands

```
# Build image
docker build -t myapp .

# Run container
docker run myapp

# Run interactively (get a shell)
docker run -it myapp /bin/bash

# Share files between host and container
docker run -v $(pwd)/data:/app/data myapp

# See running containers
docker ps

# Stop a container
```

When to Use Docker

Use Docker when:

- Sharing with others on different OS
- Deploying to cloud/servers
- Complex dependencies (CUDA, system libraries)
- Team projects

Skip Docker when:

- Personal projects on one machine
- Quick prototyping
- Simple pure-Python code

Start with venv + requirements.txt. Add Docker when needed.

Part 4: Project Structure

Organize for reproducibility

A Reproducible Project Structure

```
netflix-predictor/
├── data/
│   ├── raw/                # Original, never modified
│   └── processed/          # Cleaned data
├── models/                 # Saved models
├── notebooks/              # Jupyter notebooks
├── src/                    # Source code
│   ├── data.py            # Data loading
│   ├── train.py           # Training script
│   └── predict.py         # Prediction script
├── requirements.txt        # Dependencies
├── README.md               # Documentation
├── .gitignore             # What to ignore in Git
└── config.yaml            # Configuration
```

The README: Your Project's Front Door

Every project needs a good README:

```
# Netflix Movie Predictor

Predicts movie success based on features.

## Setup

1. Create virtual environment:
   python -m venv venv
   source venv/bin/activate

2. Install dependencies:
   pip install -r requirements.txt

3. Download data:
   python src/download_data.py

## Usage
```

Configuration Files

Don't hardcode values in your code!

```python
# Bad
learning_rate = 0.01
batch_size = 32
model_path = "/home/nipun/models/netflix.pkl"
```

Use a config file:

```yaml
# config.yaml
training:
  learning_rate: 0.01
  batch_size: 32
  epochs: 100

paths:
  model: models/netflix.pkl
  data: data/processed/
```

Loading Config Files

```python
import yaml

def load_config(path="config.yaml"):
    with open(path) as f:
        return yaml.safe_load(f)

config = load_config()
print(config["training"]["learning_rate"])  # 0.01
```

Benefits:

- Change settings without modifying code
- Track configuration in Git
- Different configs for dev/prod

.gitignore: What NOT to Track

```
# Data files (too large for Git)
data/raw/
*.csv

# Models (too large)
models/*.pkl
*.pth

# Environment
venv/
__pycache__/

# Secrets
.env
secrets.yaml
```

Part 5: Putting It Together

Reproducibility checklist

Reproducibility Checklist

Before sharing your project:

- [ ] **Virtual environment** - venv or conda
- [ ] **requirements.txt** - with pinned versions
- [ ] **Random seeds** - set at script start
- [ ] **README** - setup and usage instructions
- [ ] **Config file** - no hardcoded values
- [ ] **.gitignore** - exclude data/models
- [ ] **Test it** - clone fresh and run
- [ ] **Docker** (optional) - for complex setups

Quick Setup Script

Create `setup.sh`:

```bash
#!/bin/bash

# Create virtual environment
python -m venv venv
source venv/bin/activate

# Install dependencies
pip install -r requirements.txt

# Download data (if needed)
python src/download_data.py

echo "Setup complete! Run: source venv/bin/activate"
```

Now anyone can run: `bash setup.sh`

Netflix Project: Reproducibility

Let's apply this to our project:

```
netflix-predictor/
├── data/
│   └── movies.csv
├── src/
│   ├── train.py
│   └── predict.py
├── models/
│   └── .gitkeep
├── requirements.txt
├── config.yaml
├── README.md
├── .gitignore
└── setup.sh
```

Now anyone can reproduce our movie predictor!

Key Takeaways

1. **Virtual environments** isolate project dependencies
    - Use venv or conda
    - Pin versions in requirements.txt
2. **Random seeds** ensure reproducible results
    - Set at script start
    - Use random_state parameter
3. **Docker** packages everything (when needed)
    - OS + Python + libraries + code
4. **Project structure** matters
    - README, config, .gitignore
    - Separate code, data, models

Common Mistakes

- Not pinning versions in requirements.txt
- Forgetting random_state in train_test_split
- Committing data/models to Git (use .gitignore!)
- Hardcoding file paths ("/home/nipun/...")
- No README (how do I run this?)
- Testing only on your machine

The test: Can a friend run your code from scratch?

Lab Preview

This week's hands-on:

1. Create a virtual environment for your Netflix project
2. Generate requirements.txt with pinned versions
3. Add random seeds to your training script
4. Create a proper README
5. Write a Dockerfile (optional bonus)
6. Have a friend test your setup!

Interview Questions

Common interview questions on reproducibility:

1. **"How would you ensure your ML experiments are reproducible?"**
   - Pin all dependency versions in requirements.txt
   - Set random seeds (Python, NumPy, PyTorch/TensorFlow)
   - Version control data with DVC or similar
   - Use config files instead of hardcoded values
   - Document environment (Python version, OS)
2. **"What is Docker and why use it for ML?"**
   - Container packages code + dependencies + environment
   - "Works on my machine" → "Works everywhere"
   - Consistent dev/prod environments
   - Easy deployment and scaling

Questions?

Today's key concepts:

- Virtual environments (venv, conda)
- requirements.txt
- Random seeds
- Docker basics
- Project structure

Remember: Reproducibility is a gift to your future self!