Reproducibility & Environments

CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra, IIT Gandhinagar

The Reproducibility Crisis

"It works on my machine!"

Common scenarios:

- Research paper: "We achieved 95% accuracy"
- You: Run their code → 73% accuracy (or crash)
- Collaborator: "I can't install your dependencies"
- 6 months later: You can't run your own code

Root causes:

- Different Python versions
- Missing dependencies
- OS-specific issues
- Hardware differences

What is Reproducibility?

Reproducibility: Ability to obtain same results using same code and data

Levels:

1. **Computational Reproducibility**: Same code + data = same results
2. **Statistical Reproducibility**: Control for randomness
3. **Scientific Reproducibility**: Independent verification

Why it matters:

- Scientific integrity
- Collaboration
- Debugging
- Deployment
- Future you will thank present you

Environment Management

The Problem:

```
Project A: Python 3.8, TensorFlow 2.4
Project B: Python 3.10, TensorFlow 2.12
System: Python 3.9, TensorFlow 2.8
```

Solution: Isolated environments

Tools:

- **venv/virtualenv**: Python virtual environments
- **conda**: Package and environment manager
- **Poetry**: Modern dependency management
- **Docker**: OS-level isolation

Python Virtual Environments (venv)

Create isolated Python environments

```
# Create virtual environment
python -m venv myenv

# Activate (Linux/Mac)
source myenv/bin/activate

# Activate (Windows)
myenv\Scripts\activate

# Install packages
pip install numpy pandas scikit-learn

# Save dependencies
pip freeze > requirements.txt

# Deactivate
deactivate
```

requirements.txt

Standard way to specify dependencies

```
numpy==1.24.3
pandas==2.0.2
scikit-learn==1.2.2
matplotlib==3.7.1
torch==2.0.1
```

Install from requirements.txt:

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

Best practices:

- Pin versions (==) for reproducibility
- Use >= for flexibility
- Separate dev dependencies

Conda Environments

More powerful than venv, handles non-Python dependencies

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

# Activate
conda activate myenv

# Install packages
conda install numpy pandas scikit-learn

# Install from conda-forge
conda install -c conda-forge librosa

# Export environment
conda env export > environment.yml

# Create from YAML
conda env create -f environment.yml

# Deactivate
conda deactivate
```

environment.yml

Conda environment specification

```yaml
name: ml-project
channels:
  - pytorch
  - conda-forge
  - defaults
dependencies:
  - python=3.10
  - numpy=1.24
  - pandas=2.0
  - pytorch=2.0
  - cudatoolkit=11.8
  - pip
  - pip:
    - transformers==4.30.0
    - wandb==0.15.0
```

Advantages:

Poetry: Modern Dependency Management

Handles dependencies, virtual environments, and packaging

```
# Install Poetry
curl -sSL https://install.python-poetry.org | python3 -

# Initialize project
poetry init

# Add dependencies
poetry add numpy pandas scikit-learn

# Add dev dependencies
poetry add --group dev pytest black

# Install all dependencies
poetry install

# Run command in environment
poetry run python train.py

# Build package
poetry build
```

pyproject.toml (Poetry)

```toml
[tool.poetry]
name = "ml-project"
version = "0.1.0"
description = "My ML project"
authors = ["Your Name <you@example.com>"]

[tool.poetry.dependencies]
python = "^3.10"
numpy = "^1.24"
pandas = "^2.0"
scikit-learn = "^1.2"
torch = "^2.0"

[tool.poetry.group.dev.dependencies]
pytest = "^7.3"
black = "^23.3"
mypy = "^1.3"

[build-system]
requires = ["poetry-core"]
build-backend = "poetry.core.masonry.api"
```

Setting Random Seeds

Ensure reproducible results

```python
import random
import numpy as np
import torch

def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Make CUDA deterministic
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Set at start of script
set_seed(42)

# Now your code is reproducible
```

What is Docker?

Containerization platform: Package code + dependencies + OS

Container vs VM:

Container	Virtual Machine
Lightweight (MB)	Heavy (GB)

Benefits:

- Works identically everywhere
- No "works on my machine" problems
- Easy to share and deploy
- Reproducible environments

Docker Architecture

Key concepts:

- **Image**: Template for containers (like a class)
- **Container**: Running instance (like an object)
- **Dockerfile**: Instructions to build image
- **Registry**: Store images (Docker Hub, GitHub Container Registry)

Workflow:

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

Dockerfile Basics

Create `Dockerfile`:

```
# Start from base image
FROM python:3.10-slim

# Set working directory
WORKDIR /app

# Copy requirements
COPY requirements.txt .

# Install dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy project files
COPY . .

# Set environment variable
ENV PYTHONUNBUFFERED=1

# Run command
CMD ["python", "train.py"]
```

Building and Running Docker Images

Build image:

```
docker build -t my-ml-project:v1 .
```

Run container:

```
# Basic run
docker run my-ml-project:v1

# Interactive mode
docker run -it my-ml-project:v1 /bin/bash

# Mount volume (share files)
docker run -v $(pwd)/data:/app/data my-ml-project:v1

# Expose port (for APIs)
docker run -p 8000:8000 my-ml-project:v1

# Run with GPU
docker run --gpus all my-ml-project:v1
```

Dockerfile for ML Projects: Key Structure

Essential Dockerfile components:

```dockerfile
# 1. Base image
FROM pytorch/pytorch:2.0.1-cuda11.7-cudnn8-runtime

# 2. Working directory
WORKDIR /app

# 3. Install dependencies (cached layer)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 4. Copy application code (changes frequently)
COPY . .

# 5. Run command
CMD ["python", "train.py"]
```

Key insight: Order matters for caching! Dependencies change less than code.

Dockerfile for ML Projects: Complete Example

```dockerfile
FROM pytorch/pytorch:2.0.1-cuda11.7-cudnn8-runtime

WORKDIR /app

# System dependencies
RUN apt-get update && apt-get install -y git wget \
    && rm -rf /var/lib/apt/lists/*

# Python dependencies
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Application code
COPY . .

# Setup
RUN mkdir -p /app/data /app/models /app/outputs
ENV PYTHONPATH=/app

CMD ["python", "train.py"]
```

Docker Compose

Manage multi-container applications

Create `docker-compose.yml`:

```yaml
version: '3.8'

services:
  ml-training:
    build: .
    volumes:
      - ./data:/app/data
      - ./models:/app/models
    environment:
      - CUDA_VISIBLE_DEVICES=0
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]

  api:
    image: my-api:latest
    ports:
```

Using Docker Compose

```
# Start all services
docker-compose up

# Start in background
docker-compose up -d

# Stop services
docker-compose down

# View logs
docker-compose logs -f

# Run specific service
docker-compose run ml-training python train.py

# Rebuild images
docker-compose build
```

Multi-Stage Docker Builds: Concept

Problem: Build tools bloat the final image

Solution: Build in one stage, run in another

Benefits:

- Final image only contains runtime dependencies
- Typical savings: 500MB → 200MB (60% reduction)
- More secure (fewer tools = smaller attack surface)

Multi-Stage Docker Builds: Implementation

```dockerfile
# Stage 1: Build (includes build tools)
FROM python:3.10 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Stage 2: Runtime (slim base, no build tools)
FROM python:3.10-slim
WORKDIR /app

# Copy ONLY the installed packages, not the build tools
COPY --from=builder /root/.local /root/.local
COPY . .

ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

Key: `COPY --from=builder` pulls artifacts from stage 1

Docker Best Practices

1. Use specific base images

```dockerfile
# Good
FROM python:3.10-slim

# Avoid
FROM python:latest
```

2. Minimize layers

```dockerfile
# Good: Single RUN
RUN apt-get update && apt-get install -y \
    package1 package2 \
    && rm -rf /var/lib/apt/lists/*

# Bad: Multiple RUNs
RUN apt-get update
RUN apt-get install -y package1
RUN apt-get install -y package2
```

.dockerignore

Exclude files from Docker context

```
# .dockerignore
__pycache__
*.pyc
*.pyo
*.pyd
.Python
*.so
*.egg
*.egg-info
dist
build
.git
.gitignore
.vscode
.idea
*.ipynb_checkpoints
data/
models/*.pth
*.log
.env
```

Version Control for ML

Git + DVC (Data Version Control)

Problem: Git doesn't handle large files well

DVC: Version control for data and models

```
# Initialize DVC
dvc init

# Track large file
dvc add data/large_dataset.csv

# Commit
git add data/large_dataset.csv.dvc .gitignore
git commit -m "Add dataset"

# Push data to remote storage (S3, GCS, etc.)
dvc remote add -d myremote s3://mybucket/dvc-storage
dvc push

# Pull data
```

DVC Pipeline

Define reproducible ML pipelines

```yaml
# dvc.yaml
stages:
  prepare:
    cmd: python prepare.py
    deps:
      - prepare.py
      - data/raw
    outs:
      - data/processed

  train:
    cmd: python train.py
    deps:
      - train.py
      - data/processed
    params:
      - train.learning_rate
      - train.epochs
    outs:
      - models/model.pkl
    metrics:
      - metrics.json:
          cache: false
```

Running DVC Pipelines

```
# Reproduce pipeline
dvc repro

# Visualize pipeline
dvc dag

# Compare experiments
dvc params diff

# Track metrics
dvc metrics show

# Create experiment
dvc exp run

# Compare experiments
dvc exp diff
```

MLflow for Experiment Tracking

Track experiments, models, and parameters

```python
import mlflow
import mlflow.sklearn

# Start tracking
mlflow.set_tracking_uri("http://localhost:5000")
mlflow.set_experiment("my-experiment")

with mlflow.start_run():
    # Log parameters
    mlflow.log_param("learning_rate", 0.01)
    mlflow.log_param("epochs", 100)

    # Train model
    model = train_model()

    # Log metrics
    mlflow.log_metric("accuracy", 0.95)
    mlflow.log_metric("loss", 0.05)

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

    # Log artifacts
    mlflow.log_artifact("plots/confusion_matrix.png")
```

MLflow UI

```
# Start MLflow server
mlflow ui --host 0.0.0.0 --port 5000

# Access at http://localhost:5000
```

Features:

- Compare experiment runs
- Visualize metrics
- Download models and artifacts
- Search and filter experiments
- Deploy models

Weights & Biases (wandb)

Modern experiment tracking and collaboration

```python
import wandb

# Initialize
wandb.init(
    project="my-project",
    config={
        "learning_rate": 0.01,
        "epochs": 100,
        "batch_size": 32
    }
)

# Log metrics during training
for epoch in range(epochs):
    train_loss = train_epoch()
    val_loss = validate()

    wandb.log({
        "epoch": epoch,
        "train_loss": train_loss,
        "val_loss": val_loss
    })

# Log media
wandb.log({"confusion_matrix": wandb.plot.confusion_matrix(...)})

# Finish
wandb.finish()
```

Configuration Management

Use config files instead of hardcoding

1. YAML/JSON configs:

```yaml
# config.yaml
model:
  name: "resnet50"
  pretrained: true

training:
  batch_size: 32
  learning_rate: 0.001
  epochs: 100
  device: "cuda"

data:
  train_path: "data/train"
  val_path: "data/val"
  num_workers: 4
```

Loading Configs

```python
import yaml
from dataclasses import dataclass

@dataclass
class TrainingConfig:
    batch_size: int
    learning_rate: float
    epochs: int
    device: str

def load_config(config_path):
    with open(config_path) as f:
        config = yaml.safe_load(f)

    training_config = TrainingConfig(**config['training'])
    return training_config

# Usage
config = load_config('config.yaml')
print(config.learning_rate)
```

Hydra for Configuration

Powerful configuration management

```python
import hydra
from omegaconf import DictConfig

@hydra.main(config_path="configs", config_name="config", version_base=None)
def train(cfg: DictConfig):
    print(f"Learning rate: {cfg.training.learning_rate}")
    print(f"Batch size: {cfg.training.batch_size}")

    # Access nested config
    model = build_model(cfg.model)
    train_model(model, cfg.training)

if __name__ == "__main__":
    train()
```

Run with overrides:

```
python train.py training.learning_rate=0.01 training.epochs=50
```

Environment Variables

Store secrets and config

Create `.env` file:

```
API_KEY=your_secret_key
DATABASE_URL=postgresql://localhost/db
MODEL_PATH=/path/to/models
DEBUG=True
```

Load in Python:

```python
import os
from dotenv import load_dotenv

load_dotenv()

api_key = os.getenv("API_KEY")
db_url = os.getenv("DATABASE_URL")
debug = os.getenv("DEBUG", "False") == "True"
```

Logging Best Practices

Structured logging for debugging

```
import logging

# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('training.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# Use throughout code
logger.info("Starting training")
logger.debug(f"Batch size: {batch_size}")
logger.warning("Learning rate is very high")
logger.error("Failed to load checkpoint")

try:
    load_model()
except Exception as e:
    logger.exception("Error loading model")
```

Model Checkpointing

Save model state regularly

```
import torch

def save_checkpoint(model, optimizer, epoch, loss, path):
    checkpoint = {
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
        'config': config
    }
    torch.save(checkpoint, path)
    logger.info(f"Checkpoint saved to {path}")

def load_checkpoint(model, optimizer, path):
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    return model, optimizer, epoch, loss

# Save every N epochs
if epoch % 10 == 0:
    save_checkpoint(model, optimizer, epoch, loss,
                   f'checkpoints/model_epoch_{epoch}.pth')
```

Project Structure: Key Components

Organize your ML project for reproducibility:

Directory	Purpose
`data/`	Raw, processed, and external data
`models/`	Trained model checkpoints
`notebooks/`	Exploratory analysis
`src/`	Source code (data, models, utils)
`tests/`	Unit and integration tests

Key files: `requirements.txt`, `Dockerfile`, `README.md`, `.gitignore`

Principle: Separate code, data, config, and outputs

Project Structure: Full Layout

```
ml-project/
├── data/
│   ├── raw/              # Original, immutable data
│   ├── processed/        # Cleaned, transformed data
│   └── external/         # Data from third parties
├── models/
│   └── trained/          # Model checkpoints
├── notebooks/            # Jupyter notebooks
├── src/                  # Source code
│   ├── data/             # Data loading, preprocessing
│   ├── models/           # Model definitions
│   └── utils/            # Helper functions
├── tests/                # pytest tests
├── configs/              # YAML/JSON configs
├── requirements.txt      # Python dependencies
├── Dockerfile            # Container definition
├── README.md             # Documentation
└── .gitignore            # Ignore data/models in git
```

Reproducibility Checklist

Before sharing code:

- [ ] Pin all dependency versions
- [ ] Set random seeds
- [ ] Document Python version
- [ ] Include requirements.txt or environment.yml
- [ ] Provide Docker setup (Dockerfile)
- [ ] Document data sources
- [ ] Include sample data or download scripts
- [ ] Write clear README with setup instructions
- [ ] Test on clean environment
- [ ] Document hardware requirements (GPU, RAM)
- [ ] Include configuration files

README Template

```
# Project Name

## Setup

### Requirements
- Python 3.10+
- CUDA 11.7 (for GPU support)
- 16GB RAM minimum

### Installation

1. Clone repository
   \`\`\`bash
   git clone https://github.com/user/project.git
   cd project
   \`\`\`

2. Create virtual environment
   \`\`\`bash
   python -m venv venv
   source venv/bin/activate
   \`\`\`

3. Install dependencies
   \`\`\`bash
   pip install -r requirements.txt
   \`\`\`

4. Download data
   \`\`\`bash
   python scripts/download_data.py
   \`\`\`

## Usage

Train model:
\`\`\`bash
python train.py --config configs/config.yaml
\`\`\`

## Docker

\`\`\`bash
docker build -t myproject .
docker run -v $(pwd)/data:/app/data myproject
\`\`\`
```

Continuous Integration for ML

Automate testing and validation

`.github/workflows/train.yml`:

```
name: Train Model

on:
  push:
    branches: [ main ]

jobs:
  train:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.10

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

    - name: Run training
      run: |
        python train.py --epochs 5

    - name: Upload metrics
      uses: actions/upload-artifact@v2
```

Model Cards

Document model details

```
# Model Card: Sentiment Classifier

## Model Details
- **Model type**: DistilBERT
- **Version**: 1.0
- **Date**: 2025-12-08

## Intended Use
- **Primary use**: Sentiment analysis of product reviews
- **Out-of-scope**: Political content, medical text

## Training Data
- **Dataset**: Amazon reviews (100k samples)
- **Splits**: 80/10/10 (train/val/test)
- **Preprocessing**: Lowercase, remove URLs

## Performance
- **Test Accuracy**: 92%
- **F1 Score**: 0.91

## Limitations
- Struggles with sarcasm
- Biased toward longer reviews

## Ethical Considerations
- May amplify existing biases in training data
```

What We've Learned

Environment Management:

- venv, conda, Poetry for Python environments
- requirements.txt, environment.yml, pyproject.toml

Containerization:

- Docker for reproducible environments
- Dockerfile, docker-compose
- Multi-stage builds

Version Control:

- Git for code
- DVC for data and models

Deterministic Training in PyTorch

Problem: GPUs use non-deterministic algorithms by default.

Full Determinism:

```
import torch
import numpy as np
import random

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

    # Deterministic algorithms
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

    # For CUDA >= 10.2
    torch.use_deterministic_algorithms(True)
```

Handling Non-Deterministic Operations

Some operations are inherently non-deterministic on GPU:

Atomic adds (scatter, index_add):

```
# Solution: Use deterministic scatter
torch.use_deterministic_algorithms(True, warn_only=True)
```

Data loading randomness:

```
def worker_init_fn(worker_id):
    """Set seed per worker."""
    np.random.seed(42 + worker_id)
    random.seed(42 + worker_id)

dataloader = DataLoader(
    dataset,
    batch_size=32,
    shuffle=True,
    worker_init_fn=worker_init_fn,
    generator=torch.Generator().manual_seed(42)
```

Hash-Based Reproducibility

Content-Addressed Storage: Use hash of data/code, not timestamps.

DVC uses content addressing:

```
# Data is identified by MD5 hash
$ dvc add data/train.csv
# Creates: data/train.csv.dvc with hash

# Track changes
$ git add data/train.csv.dvc
$ git commit -m "Update training data"

# Anyone can retrieve exact version
$ dvc pull
```

Benefits:

- Detect silent data changes
- Ensure exact data version

Semantic Versioning for ML

Code versioning: Git tags (v1.2.3)

Data versioning: DVC tags

Model versioning:

```python
# model_registry.py
class ModelVersion:
    def __init__(self, major, minor, patch):
        self.version = f"{major}.{minor}.{patch}"
        self.code_commit = get_git_commit()
        self.data_version = get_dvc_version()
        self.config = load_config()
        self.metrics = {}

    def save(self, path):
        metadata = {
            'version': self.version,
            'code_commit': self.code_commit,
            'data_version': self.data_version,
            'config': self.config,
            'metrics': self.metrics
```

Dependency Resolution Best Practices

Problem: Dependency hell.

pip-tools for locked dependencies:

```
# requirements.in (high-level)
numpy
pandas
scikit-learn

# Generate locked requirements.txt
pip-compile requirements.in

# Result: requirements.txt with all transitive deps pinned
numpy==1.24.3
pandas==2.0.2
  # via -r requirements.in
scikit-learn==1.2.2
  # via -r requirements.in
scipy==1.10.1
  # via scikit-learn
```

Container Best Practices

Multi-stage builds (smaller images):

```
# Build stage
FROM python:3.10 AS builder
WORKDIR /app
COPY requirements.txt .
RUN pip install --user -r requirements.txt

# Runtime stage
FROM python:3.10-slim
WORKDIR /app
COPY --from=builder /root/.local /root/.local
COPY . .
ENV PATH=/root/.local/bin:$PATH
CMD ["python", "app.py"]
```

Benefits: Final image ~200MB vs ~1GB.

Security:

Testing for Reproducibility

Automated reproducibility tests:

```python
def test_reproducibility():
    """Test that model training is deterministic."""
    set_seed(42)
    model1 = train_model(data, epochs=5)
    loss1 = model1.evaluate(test_data)

    set_seed(42)
    model2 = train_model(data, epochs=5)
    loss2 = model2.evaluate(test_data)

    # Should be EXACTLY equal
    assert loss1 == loss2, f"Non-deterministic: {loss1} != {loss2}"

    # Check model weights are identical
    for p1, p2 in zip(model1.parameters(), model2.parameters()):
        assert torch.allclose(p1, p2), "Weights differ!"
```

Run in CI/CD to catch non-determinism early.

Continuous Reproducibility Monitoring

Track reproducibility metrics over time:

```python
def log_reproducibility_metadata():
    """Log all factors affecting reproducibility."""
    metadata = {
        'git_commit': get_git_hash(),
        'git_branch': get_git_branch(),
        'data_hash': compute_data_hash(),
        'python_version': sys.version,
        'torch_version': torch.__version__,
        'cuda_version': torch.version.cuda,
        'cudnn_version': torch.backends.cudnn.version(),
        'hostname': socket.gethostname(),
        'timestamp': datetime.now().isoformat(),
        'random_seed': 42,
        'config': load_config()
    }

    # Log to MLflow/W&B
    mlflow.log_params(metadata)

    return metadata
```

Reproducibility Checklist

Before running experiments:

- [ ] Set all random seeds
- [ ] Pin all package versions
- [ ] Document Python version
- [ ] Version control code (Git)
- [ ] Version control data (DVC)
- [ ] Document hardware (GPU model, CUDA version)

During training:

- [ ] Log hyperparameters
- [ ] Save checkpoints with metadata
- [ ] Track metrics

Resources

Tools:

- Docker: https://docs.docker.com/
- Poetry: https://python-poetry.org/
- DVC: https://dvc.org/
- MLflow: https://mlflow.org/
- Weights & Biases: https://wandb.ai/

Guides:

- Reproducible ML: https://www.nature.com/articles/s41592-021-01256-7
- Docker for Data Science: https://towardsdatascience.com/docker-for-data-science
- DVC Tutorial: https://dvc.org/doc/start

Questions?

Next: Testing & CI/CD for AI

Lab: Dockerize ML project, setup experiment tracking