STAT 4160: Data Science Productivity Tools

Yi Wang

2025-08-13

Table of contents

Preface

This is the lecture note for the course STAT 4160.

Introduction

13-week plan (2 × 75-min per week)

Week 1 – Setup, Colab, Git/GitHub

- *Lec A:* Local Python + VS Code; Colab basics (GPU, Drive mount, persistence limits), repo cloning in Colab, `requirements.txt`, seeds.
- *Lec B:* Git essentials, branching, PRs, code review etiquette, `.gitignore`, **Git-LFS** do's/don'ts (quota pitfalls).
- **Deliverable:** Team repo with a Colab notebook that runs and logs environment info; one PR merged.

Week 2 – Reproducible reporting (Quarto) + RStudio cameo

- *Lec A:* Quarto for Python: parameters, caching, citations; publish to GitHub Pages.
- *Lec B (15–25 min cameo):* RStudio + Quarto rendering (so they can read R-centric docs later), then back to Python.
- **Deliverable:** Parameterized EDA report (symbol, date range as params).

Week 3 – Unix for data work + automation

- *Lec A:* Shell basics (pipes, redirects), `grep/sed/awk`, `find/xargs`, regex.
- *Lec B:* Shell scripts, simple **Makefile/justfile** targets; `rsync`, quick SSH/tmux tour.
- **Deliverable:** `make get-data` and `make report` run end-to-end.

Week 4 – SQL I (schemas, joins)

- *Lec A:* SQLite in repo; schema design for OHLCV + metadata; SELECT/JOIN/GROUP BY.
- *Lec B:* Window functions; indices; `pandas.read_sql` pipelines.
- **Deliverable:** SQL notebook producing a tidy table ready for modeling.

Week 5 – pandas for time series

- *Lec A:* Cleaning, types, missing, merges; `groupby`, pivot; Parquet I/O.

- *Lec B:* Time-series ops: resampling, rolling windows, shifting/lagging, calendar effects.
- **Deliverable:** Cleaned Parquet dataset + feature snapshot.

Week 6 – APIs & Web scraping (ethics + caching)

- *Lec A:* HTTP basics, `requests`, pagination, auth, retries, backoff; **don't hard-code keys** (`python-dotenv`).
- *Lec B:* BeautifulSoup, CSS selectors, robots.txt, throttling; **cache raw pulls**; persist to SQL/Parquet.
- **Deliverable:** One external data source ingested with caching & schema checks.

Week 7 – Quality: tests, lint, minimal CI

- *Lec A:* `pytest` (2–3 meaningful tests), data validation (light Pandera or custom checks), logging, type hints.
- *Lec B:* **Pre-commit** (black, ruff, nbstripout), **GitHub Actions** to run tests + lint on PRs (fast jobs only).
- **Deliverable:** CI badge green; failing test demonstrates leakage prevention or schema guard.

Week 8 – Time-series baselines & backtesting

- *Lec A:* Problem framing; horizon, step size; MAE/sMAPE/MASE; **rolling-origin** evaluation.
- *Lec B:* Baselines: naive/seasonal-naive; quick ARIMA/Prophet or sklearn regressor with lags.
- **Deliverable:** Baseline model card + backtest plot in Quarto.

Week 9 – Finance-specific evaluation & leakage control

- *Lec A:* **Feature timing & label definition** (t+1 returns, multi-step horizons), survivorship bias, look-ahead traps, data snooping.
- *Lec B:* **Walk-forward / expanding window**, embargoed splits, drift detection; error analysis by regime (volatility bins, bull/bear).
- **Deliverable:** A robust evaluation plan + revised splits; leakage test added to `pytest`.

Week 10 – PyTorch fundamentals

- *Lec A:* Tensors, autograd, datasets/dataloaders for windows; training loop, early stopping; GPU in Colab; mixed precision.
- *Lec B:* A small **LSTM/TCN** baseline for forecasting; monitoring loss/metrics; save best weights.
- **Deliverable:** PyTorch baseline surpasses classical baseline on at least one metric.

Week 11 – Transformers for sequences (tiny GPT)

- *Lec A:* Attention from scratch; **tiny char-level GPT** (embeddings, positions, single head → multi-head), sanity-check overfitting on toy data.
- *Lec B:* Adapt to time series: window embedding, causal masking, regression head; ablation (context length, heads, dropout) within Colab budget.
- **Deliverable:** Transformer results + one ablation figure; notes on compute/time.

Week 12 – Productivity at scale (lightweight)

- *Lec A:* Packaging a small library (`src/` layout, `pyproject.toml`), simple **CLI** (Typer) for batch inference; config via YAML.
- *Lec B:* **Optional FastAPI** endpoint demo (local only) + reproducibility audit (fresh-clone run).
- **Deliverable:** Tagged release `v1.0-rc`, CLI can score a held-out period and write a report.

Week 13 – Communication & showcase

- *Lec A:* Poster + abstract workshop; tell the error-analysis story; figure polish; README & model card.
- *Lec B:* In-class presentations + final feedback; plan for continuing to the Spring symposium (next-steps backlog).
- **Deliverable:** Poster draft, 250-word abstract, and a reproducible repo ready to extend.

Project spine

- **Milestones:** W1 repo & env → W3 automated data pipeline → W6 external data → W7 CI green → W8 baselines → W9 robust eval plan → W10 PyTorch baseline → W11 tiny Transformer → W12 release candidate → W13 poster & talk.
- **Tracking (minimal):** log experiments to a simple CSV (`results/experiments.csv`) and keep a Quarto "lab notebook." If you're open to one extra tool, free **Weights & Biases** makes ablations much easier—but the CSV+Quarto path is fine for two students.
- **Data strategy:** keep raw data out of Git (use `make get-data`); store processed Parquet under 100MB if you must commit; otherwise regenerate. Use Git-LFS only for small, immutable artifacts to avoid quota pain.
- **Secrets:** `.env` with `python-dotenv` + `.env` in `.gitignore`. For Colab, use environment variables or a JSON in Drive (not committed).

1 Session 1 — Dev environment & Colab workflow

1.1 Session 1 — Dev environment & Colab workflow

1.1.1 Learning goals

By the end of class, students can:

1. Mount Google Drive in Colab and work in a persistent course folder.
2. Clone a GitHub repo into Drive (or create a project folder if no repo yet).
3. Create and install from a **soft-pinned** `requirements.txt`.
4. Verify **environment info** (Python, OS, library versions) and **GPU availability**.
5. Use a **reproducibility seed** pattern (NumPy + PyTorch) and validate it.
6. Save a simple **system check report** to the repo.

1.2 Agenda (75 min)

- **(5 min)** Course framing: how we'll work this semester
- **(12 min)** Slides & demo: Colab + Drive persistence; project folders; soft vs hard pins
- **(8 min)** Slides & demo: reproducibility basics (seeds, RNG, deterministic ops)
- **(35 min) In-class lab** (Colab): mount Drive → clone/create project → requirements → environment check → reproducibility check → write report
- **(10 min)** Wrap-up, troubleshooting, and homework briefing

---

1.3 Main Points

Why Colab + Drive

- Colab gives you GPUs and a clean Python every session.
- The runtime is **ephemeral**. Anything under `/content` disappears.
- Mount **Drive** and work under `/content/drive/MyDrive/...` to persist code and outputs.

Project layout (today's minimal)

```
project/
  reports/
  notebooks/
  data/
  requirements.txt
  system_check.ipynb
```

(We'll add `src/`, tests, CI in later sessions.)

Pins: soft vs hard

- **Soft pins** (e.g., `pandas>=2.2,<3.0`) keep you compatible across machines.
- **Hard pins** (exact versions) are for releases. Today we'll use **soft pins**, then **freeze** to `requirements-lock.txt` in homework.

Reproducibility basics

- Fix seeds for **random**, **NumPy**, **PyTorch** (and CUDA if present).
- Disable nondeterministic cuDNN behavior for repeatability in simple models.
- **Beware**: some ops remain nondeterministic on GPU; we'll use simple ones.

Minimal Git today

- If you already have a repo: clone it into G-Drive.
- If not: create a folder; later you can upload the notebook via GitHub web UI.
- Full Git workflow (branch/PR/CI) starts next session.

1.4 In-class Lab (35 min)

Instructor tip: Put these as sequential Colab cells. Students should run them top-to-bottom. Replace placeholders like `YOUR_USERNAME` / `YOUR_REPO` before class if you already created a starter repo. If not, tell them to use the "no-repo" path in Step 3B.

1.4.1 1) Mount Google Drive and create a course folder

```
# Colab cell
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

COURSE_DIR = "/content/drive/MyDrive/dspt25"  # change if you prefer another path
PROJECT_NAME = "unified-stocks"               # course project folder/repo name
```

Save it as `system_check.ipynb`.

```
# Colab cell: make directories and cd into project folder
import os, pathlib
base = pathlib.Path(COURSE_DIR)
proj = base / PROJECT_NAME  # / is overloaded to create the path
for p in [base, proj, proj/"reports", proj/"notebooks", proj/"data"]:
    p.mkdir(parents=True, exist_ok=True)

import os
os.chdir(proj)
print("Working in:", os.getcwd())
```

1.4.2 2) (Optional) If you already have a GitHub repo, clone it into Drive

Pick A or B (not both).

A. Clone an existing repo (recommended if you created a starter repo)

```
# Colab cell: clone via HTTPS (public or your private; for private, you can upload later ins
REPO_URL = "https://github.com/YOUR_ORG_OR_USERNAME/YOUR_REPO.git"  # <- change me
import subprocess, os
os.chdir(base)  # clone next to your project folder
```

```
subprocess.run(["git", "clone", REPO_URL], check=True) # check if there is an error inseat o
# Optionally, use that cloned repo as the working directory:(uncomment the lines below if do
# REPO_NAME = REPO_URL.split("/")[-1].replace(".git","")
# os.chdir(base/REPO_NAME)
# print("Working in:", os.getcwd())
os.chdir(proj) # change back to proj dir
print("Working in:", os.getcwd())
```

B. No repo yet? Stay with the folder we created. You'll upload files via GitHub web UI after class.

1.4.3 3) Create a soft-pinned `requirements.txt` and install

```
# Colab cell: write a soft-pinned requirements.txt
req = """\
pandas>=2.2,<3.0
numpy>=2.0.0,<3.0
pyarrow>=15,<17
matplotlib>=3.8,<4.0
scikit-learn>=1.6,<2.0
yfinance>=0.2,<0.3
python-dotenv>=1.0,<2.0
"""
open("requirements.txt","w").write(req)
print(open("requirements.txt").read())
```

```
# Colab cell: install (quietly). Torch is usually preinstalled in Colab; we'll check separat
!pip install -q -r requirements.txt
```

```
# Colab cell: PyTorch check. If not available (rare in Colab), install CPU-only as a fallbac
try:
    import torch
    print("PyTorch:", torch.__version__)
except Exception as e:
    print("PyTorch not found; installing CPU-only wheel as fallback...")
    !pip install -q torch
    import torch
    print("PyTorch:", torch.__version__)
```

1.4.4 4) Environment report (Python/OS/lib versions, GPU availability)

```
# Colab cell: environment info + GPU check
import sys, platform, json, time
import pandas as pd
import numpy as np

env = {
    "timestamp": time.strftime("%Y-%m-%d %H:%M:%S"),
    "python": sys.version,
    "os": platform.platform(),
    "pandas": pd.__version__,
    "numpy": np.__version__,
}

try:
    import torch
    env["torch"] = torch.__version__
    env["cuda_available"] = bool(torch.cuda.is_available())
    env["cuda_device"] = torch.cuda.get_device_name(0) if torch.cuda.is_available() else "CPU
except Exception as e:
    env["torch"] = "not importable"
    env["cuda_available"] = False
    env["cuda_device"] = "CPU"

print(env)
os.makedirs("reports", exist_ok=True)
with open("reports/environment.json","w") as f:
    json.dump(env, f, indent=2)
```

1.4.5 5) Reproducibility seed utility + quick validation

```
# Colab cell: reproducibility helpers
import random
import numpy as np

def set_seed(seed: int = 42, deterministic_torch: bool = True):
    random.seed(seed)
    np.random.seed(seed)
    try:
```

```
        import torch
        torch.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
        if deterministic_torch:
            torch.backends.cudnn.deterministic = True
            torch.backends.cudnn.benchmark = False
            try:
                torch.use_deterministic_algorithms(True)
            except Exception:
                pass
    except Exception:
        pass

def sample_rng_fingerprint(n=5, seed=42):
    set_seed(seed)
    a = np.random.rand(n).round(6).tolist()
    try:
        import torch
        b = torch.rand(n).tolist()
        b = [round(x,6) for x in b]
    except Exception:
        b = ["torch-missing"]*n
    return {"numpy": a, "torch": b}

f1 = sample_rng_fingerprint(n=6, seed=123)
f2 = sample_rng_fingerprint(n=6, seed=123)
print("Fingerprint #1:", f1)
print("Fingerprint #2:", f2)
print("Match:", f1 == f2)

with open("reports/seed_fingerprint.json","w") as f:
    json.dump({"f1": f1, "f2": f2, "match": f1==f2}, f, indent=2)
```

1.4.6 6) Create (or verify) `tickers_25.csv` for the course

```
# Colab cell: create stock list if it doesn't exist yet
import pandas as pd, os
tickers = [
    "AAPL","MSFT","AMZN","GOOGL","META","NVDA","TSLA","JPM","JNJ","V",
    "PG","HD","BAC","XOM","CVX","PFE","KO","DIS","NFLX","INTC",
```

```
    "CSCO","ORCL","T","VZ","WMT"
]
path = "tickers_25.csv"
if not os.path.exists(path):
    pd.DataFrame({"ticker": tickers}).to_csv(path, index=False)
pd.read_csv(path).head()
```

1.4.7 7) (Optional) Prove GPU works by allocating a small tensor

```
# Colab cell: tiny GPU smoke test (safe if CUDA available)
import torch, time

# change back to not use deterministic_algorithm to do the matrix computation
# torch.use_deterministic_algorithms(False)

device = "cuda" if torch.cuda.is_available() else "cpu"
x = torch.randn(1000, 1000, device=device)
y = x @ x.T
print("Device:", device, "| y shape:", y.shape, "| mean:", y.float().mean().item())
```

1.4.8 8) Save a short Markdown environment report

```
# Colab cell: write a small Markdown summary for humans
from textwrap import dedent
summary = dedent(f"""
# System Check

- Timestamp: {env['timestamp']}
- Python: `{env['python']}`
- OS: `{env['os']}`
- pandas: `{env['pandas']}` | numpy: `{env['numpy']}` | torch: `{env['torch']}`
- CUDA available: `{env['cuda_available']}` | Device: `{env['cuda_device']}`

## RNG Fingerprint
- Match on repeated seeds: `{f1 == f2}`
- numpy: `{f1['numpy']}`
- torch: `{f1['torch']}`
""").strip()
```

```
open("reports/system_check.md","w").write(summary)
print(summary)
```

1.4.9 Save the file as `system_check.ipynb`. To do it automatically, you can use the following code:

```
# Colab cell: save this notebook as system_check.ipynb
from google.colab import  _message
notebook_name = "system_check.ipynb"
# Create the 'notebooks' subdirectory path
out_dir = proj / "notebooks"
out_path = out_dir / notebook_name

# Make sure the folder exists
out_dir.mkdir(parents=True, exist_ok=True)

# Get the CURRENT notebook JSON from Colab
resp = _message.blocking_request('get_ipynb', timeout_sec=10)
nb = resp.get('ipynb') if isinstance(resp, dict) else None

# Basic sanity check: ensure there are cells
if not nb or not isinstance(nb, dict) or not nb.get('cells'):
    raise RuntimeError("Could not capture the current notebook contents (no cells returned).
                       "Try running this cell again after a quick edit, or use File → Save a

# Write to Drive
with open(out_path, 'w', encoding='utf-8') as f:
    json.dump(nb, f, ensure_ascii=False, indent=2)

print("Saved notebook to:", out_path)
```

What to submit after class (if you already have a GitHub repo): For today, students may **upload** `system_check.ipynb`, `reports/environment.json`, and `reports/system_check.md` via the GitHub web UI (Add file → Upload files). We'll do proper pushes/PRs next session.

1.5 Troubleshooting notes (share in class)

- **Drive won't mount**: Refresh the Colab tab, run the mount cell again, re-authorize Google permissions.
- **`pip install` hangs**: Rerun; if it persists, restart runtime (Runtime → Restart session) and re-run from the top.
- **PyTorch mismatch**: If Colab has Torch preinstalled, don't upgrade it. If you installed a CPU wheel by mistake and want GPU later, it's usually easiest to **restart runtime**.
- **Path confusion**: Print `os.getcwd()` often; ensure you're inside your project folder under `/content/drive/MyDrive/...`.

1.6 Homework (due before Session 2)

Goal: Produce a **reproducible system snapshot** and a **seed-verified mini experiment**, then upload to your repo (via GitHub web UI if you're not comfortable pushing yet).

1.6.1 Part A — Freeze your environment

1. From the same Colab runtime (after installing), create a lock file:

   ```
   # Colab cell: freeze exact versions
   !pip freeze > requirements-lock.txt
   print("Wrote requirements-lock.txt with exact versions")
   !head -n 20 requirements-lock.txt
   ```

2. Add a note to `README.md` explaining the difference between:

   - `requirements.txt` (soft pins for development) and
   - `requirements-lock.txt` (exact versions used **today**).

1.6.2 Part B — Reproducibility mini-experiment

Create `notebooks/reproducibility_demo.ipynb` with the following cells (students copy/paste):

1) Setup & data generation

```python
import numpy as np, torch, random, json, os, time

def set_seed(seed=123):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    try:
        torch.use_deterministic_algorithms(True)
    except Exception:
        pass

def make_toy(n=512, d=10, noise=0.1, seed=123):
    set_seed(seed)
    X = torch.randn(n, d)
    true_w = torch.randn(d, 1)
    y = X @ true_w + noise * torch.randn(n, 1)
    return X, y, true_w

device = "cuda" if torch.cuda.is_available() else "cpu"
X, y, true_w = make_toy()
X, y = X.to(device), y.to(device)
```

2) Minimal training loop (linear model)

```python
def train_once(lr=0.05, steps=300, seed=123):
    set_seed(seed)
    model = torch.nn.Linear(X.shape[1], 1, bias=False).to(device)
    opt = torch.optim.SGD(model.parameters(), lr=lr)
    loss_fn = torch.nn.MSELoss()
    losses=[]
    for t in range(steps):
        opt.zero_grad(set_to_none=True)
        yhat = model(X)
        loss = loss_fn(yhat, y)
        loss.backward()
        opt.step()
        losses.append(loss.item())
    return model.weight.detach().cpu().numpy(), losses[-1]
```

```
w1, final_loss1 = train_once(seed=2025)
w2, final_loss2 = train_once(seed=2025)

print("Final loss 1:", round(final_loss1, 6))
print("Final loss 2:", round(final_loss2, 6))
print("Weights equal:", np.allclose(w1, w2, atol=1e-7))
```

3) Save results JSON

```
os.makedirs("reports", exist_ok=True)
result = {
    "device": device,
    "final_loss1": float(final_loss1),
    "final_loss2": float(final_loss2),
    "weights_equal": bool(np.allclose(w1, w2, atol=1e-7)),
    "timestamp": time.strftime("%Y-%m-%d %H:%M:%S")
}
with open("reports/reproducibility_results.json","w") as f:
    json.dump(result, f, indent=2)
result
```

Expected outcome: the two runs with the same seed should produce the **same final loss** and **identical weights** (within tolerance). If on GPU, deterministic settings should keep this stable for this simple model.

1.6.3 Part C — Add a `.env.example`

Create a placeholder for API keys we'll use later:

```
env_example = """\
# Example environment variables (do NOT commit a real .env with secrets)
ALPHA_VANTAGE_KEY=
FRED_API_KEY=
"""
open(".env.example", "w").write(env_example)
print(open(".env.example").read())
```

1.6.4 Part D — Upload to GitHub

Until we set up pushes/PRs next class, use the GitHub web UI:

- Upload: `system_check.ipynb`, `reports/environment.json`, `reports/system_check.md`, `requirements.txt`, `requirements-lock.txt`, `notebooks/reproducibility_demo.ipynb`, `reports/reproducibility_results.json`, `.env.example`.
- If you already cloned a repo in class and are comfortable pushing, you may push from your laptop instead. **Do not paste tokens into notebooks.**

1.6.5 Grading (pass/revise)

- `requirements.txt` present; `requirements-lock.txt` present and non-empty.
- `system_check.ipynb` runs and writes `reports/system_check.md` + `environment.json`.
- `reproducibility_demo.ipynb` demonstrates identical results across repeated runs with same seed and writes `reports/reproducibility_results.json`.
- `.env.example` present with placeholders.

1.7 What to emphasize

- "Colab is **ephemeral**; persist to **Drive**."
- "Soft pins now; **freeze** later."
- "Seeds are necessary but not sufficient—watch for nondeterministic ops."
- "Never store secrets (API keys) in the repo; use `.env` and keep a `.env.example`."

That's it for Session 1. In Session 2 we'll set up **Git basics and Git-LFS** and move from uploading via web UI to **branch/PR** workflows.

2 Session 2 — Git essentials & Git-LFS

> **Security note:** Today we'll push from Colab using a **short-lived GitHub personal access token (PAT)** entered interactively. **Never** hard-code or commit tokens.

2.1 Session 2 — Git essentials & Git-LFS (75 min)

2.1.1 Learning goals

By the end of class, students can:

1. Explain Git's mental model: working directory → staging → commit; branches and remotes.
2. Create a feature branch, commit changes, and push to GitHub from Colab safely.
3. Use **.gitignore** to avoid committing generated artifacts and secrets.
4. Install and configure **Git-LFS**, track large/binary files, and verify tracking.
5. Open a **pull request (PR)** and follow review etiquette.

2.2 Agenda (75 minutes)

- **(8 min)** Recap & goals; overview of today's workflow
- **(12 min)** Slides: Git mental model; branches; remotes; commit hygiene
- **(10 min)** Slides: `.gitignore` must-haves; Git-LFS (when/why); LFS quotas & pitfalls
- **(35 min) In-class lab**: clone → config → branch → `.gitignore` → LFS → sample Parquet → push → PR
- **(10 min)** Wrap-up; troubleshooting; homework briefing

2.3 Main points

2.3.1 Git mental model

- **Working directory** (your files) → `git add` → **staging** → `git commit` → **local history**
- **Remote**: GitHub hosts a copy. `git push` publishes commits; `git pull` brings others' changes.
- **Branch**: a movable pointer to a chain of commits. Default is `main`. Create **feature branches** for each change.

2.3.2 Branch & PR etiquette

- One feature/change per branch (small, reviewable diffs).
- Commit messages: *imperative mood*, short subject line (72 chars), details in body if needed:
  - `feat: add git-lfs tracking for parquet`
  - `docs: add README section on setup`
  - `chore: ignore raw data directory`
- PR description: what/why, testing notes, checklist. Tag your teammate for review.

2.3.3 `.gitignore` must-haves

- **Secrets**: `.env`, API keys (never commit).
- **Large/derived artifacts**: raw/interim data, logs, cache, compiled assets.
- **Notebooks' checkpoints**: `.ipynb_checkpoints/`.
- OS/editor cruft: `.DS_Store`, `Thumbs.db`, `.vscode/`.

2.3.4 Git-LFS

- Git-LFS = Large File Storage. Keeps **pointers** in Git; binaries in LFS storage.
- Track only what's necessary to version (e.g., *small* processed Parquet samples, posters/PDFs, small models).
- **Do not** LFS huge raw data you can re-download (`make get-data`).
- Quotas apply on Git hosting—be selective.

2.3.5 Safe pushes from Colab

- Use a **fine-grained PAT** limited to a single repo with **Contents: Read/Write** + **Pull requests: Read/Write**.
- Enter token via `getpass` (not stored). Push using a **temporary URL** (token not saved in `git config`).
- After push, **clear cell output**.

2.4 In-class lab (35 min)

> **Instructor tip:** Students should have created a repo on GitHub before this lab (e.g., `unified-stocks-teamX`). If not, give them 3 minutes to do so and add their partner as a collaborator.

We'll:

1. Mount Drive & clone the repo.
2. Configure Git identity.
3. Create a feature branch.
4. Add `.gitignore`.
5. Install and configure **Git-LFS**.
6. Track Parquet & DB files; generate a **sample Parquet**.
7. Commit & **push from Colab** using a short-lived PAT.
8. Open a PR (via web UI, optional API snippet included).

2.4.1 0) Mount Google Drive and set variables

```python
# Colab cell
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

# Adjust these two for YOUR repo
REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"
REPO_NAME  = "unified-stocks-teamX"   # e.g., unified-stocks-team1

BASE_DIR   = "/content/drive/MyDrive/dspt25"
CLONE_DIR  = f"{BASE_DIR}/{REPO_NAME}"
REPO_URL   = f"https://github.com/{REPO_OWNER}/{REPO_NAME}.git"
```

```python
import os, pathlib
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)
```

2.4.2 1) Clone the repo (or pull latest if already cloned)

```python
import os, subprocess, shutil, pathlib

if not pathlib.Path(CLONE_DIR).exists():
    !git clone {REPO_URL} {CLONE_DIR}
else:
    # If the folder exists, just ensure it's a git repo and pull latest
    os.chdir(CLONE_DIR)
    !git status
    !git pull --ff-only
os.chdir(CLONE_DIR)
print("Working dir:", os.getcwd())
```

2.4.3 2) Configure Git identity (local to this repo)

```python
# Replace with your name and school email
!git config user.name "Your Name"
!git config user.email "you@example.edu"

!git config --get user.name
!git config --get user.email
```

2.4.4 3) Create and switch to a feature branch

```python
BRANCH = "setup/git-lfs"
!git checkout -b {BRANCH}
!git branch --show-current
```

2.4.5 4) Add a robust `.gitignore`

```
gitignore = """\
# Byte-compiled / cache
__pycache__/
*.py[cod]

# Jupyter checkpoints
.ipynb_checkpoints/

# OS/editor files
.DS_Store
Thumbs.db
.vscode/

# Environments & secrets
.env
.env.*
.venv/
*.pem
*.key

# Data (raw & interim never committed)
data/raw/
data/interim/

# Logs & caches
logs/
.cache/
"""
open(".gitignore", "w").write(gitignore)
print(open(".gitignore").read())
```

2.4.6 5) Install and initialize Git-LFS (Colab)

```
# Install git-lfs on the Colab VM (one-time per runtime)
!apt-get -y update >/dev/null
!apt-get -y install git-lfs >/dev/null
!git lfs install
!git lfs version
```

2.4.7 6) Track Parquet/DB/PDF/model binaries with LFS

```
# Add .gitattributes entries via git lfs track
!git lfs track "data/processed/*.parquet"
!git lfs track "data/*.db"
!git lfs track "models/*.pt"
!git lfs track "reports/*.pdf"

# Show what LFS is tracking and verify .gitattributes created
!git lfs track
print("\n.gitattributes:")
print(open(".gitattributes").read())
```

> **Why not LFS for raw?** Raw data should be **re-downloadable** with `make get-data` later; don't burn LFS quota.

2.4.8 7) Create a small Parquet file to test LFS

```
import pandas as pd, numpy as np, os, pathlib

pathlib.Path("data/processed").mkdir(parents=True, exist_ok=True)

tickers = pd.read_csv("tickers_25.csv")["ticker"].tolist() if os.path.exists("tickers_25.csv"
    "AAPL","MSFT","AMZN","GOOGL","META","NVDA","TSLA","JPM","JNJ","V",
    "PG","HD","BAC","XOM","CVX","PFE","KO","DIS","NFLX","INTC","CSCO","ORCL","T","VZ","WMT"
]

# 1000 business days x up to 25 tickers ~ 25k rows; a few MB as Parquet
dates = pd.bdate_range("2018-01-01", periods=1000)
df = (pd.MultiIndex.from_product([tickers, dates], names=["ticker","date"])
      .to_frame(index=False))
rng = np.random.default_rng(42)
df["r_1d"] = rng.normal(0, 0.01, size=len(df))  # synthetic daily returns
df.to_parquet("data/processed/sample_returns.parquet", index=False)
df.head()
```

2.4.9 8) Stage and commit changes

```
!git add .gitignore .gitattributes data/processed/sample_returns.parquet
!git status

!git commit -m "feat: add .gitignore and git-lfs tracking; add sample Parquet"
!git log --oneline -n 2
```

2.4.10 9) Push from Colab with a short-lived token (safe method)

Create a **fine-grained PAT** at GitHub → Settings → Developer settings → **Fine-grained tokens**

- Resource owner: your username/org
- Repositories: select **this repo only**
- Permissions: **Contents (Read/Write)**, **Pull requests (Read/Write)**
- Expiration: short (e.g., 7 days)

```
# Colab cell: push using a temporary URL with token (not saved to git config)
from getpass import getpass
token = getpass("Enter your GitHub token (input hidden; not stored): ")

push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
!git push {push_url} {BRANCH}:{BRANCH}

# Optional: immediately clear the token variable
del token
```

If the command prints the URL, **clear this cell's output** after a successful push (Colab: " " → "Clear output").

2.4.11 10) Open a Pull Request

- **Recommended (web UI):** Navigate to your repo on GitHub → Compare & pull request → base: `main`, compare: `setup/git-lfs`. Fill title/description, tag your partner, and create the PR.
- **Optional (API):** open a PR programmatically from Colab:

```python
# OPTIONAL: Create PR via GitHub API (requires token again)
from getpass import getpass
import requests, json

token = getpass("GitHub token (again, not stored): ")
headers = {"Authorization": f"Bearer {token}",
           "Accept": "application/vnd.github+json"}
payload = {
    "title": "Setup: .gitignore + Git-LFS + sample Parquet",
    "head": BRANCH,
    "base": "main",
    "body": "Adds .gitignore, configures Git-LFS for parquet/db/pdf/model files, and commits
}
r = requests.post(f"https://api.github.com/repos/{REPO_OWNER}/{REPO_NAME}/pulls",
                  headers=headers, data=json.dumps(payload))
print("PR status:", r.status_code)
try:
    pr_url = r.json()["html_url"]
    print("PR URL:", pr_url)
except Exception as e:
    print("Response:", r.text)
del token
```

2.4.12 11) Quick verification checklist

- `git lfs ls-files` shows `data/processed/sample_returns.parquet`:

```
!git lfs ls-files
```

- PR diff shows a small **pointer** for the Parquet, not raw binary content.
- `.gitignore` present; no secrets or raw data committed.

2.5 Wrap-up (talking points, 10 min)

- Keep PRs small and focused; write helpful titles and descriptions.
- Don't commit secrets or large data. Use `.env` + `.env.example`.
- Use LFS *selectively*—version only small, important binaries (e.g., sample processed sets, posters).

- Next time: **Quarto** polish (already started) and **Unix** automation to fetch raw data reproducibly.

2.6 Homework (due before Session 3)

Goal: Cement branch/PR hygiene, add review scaffolding, and add a small guard against large files accidentally committed outside LFS.

2.6.1 Part A — Add a PR template and CODEOWNERS

Create a PR template so every PR includes key info.

```
# Run in your repo root
import os, pathlib, textwrap
pathlib.Path(".github").mkdir(exist_ok=True)
tpl = textwrap.dedent("""\
    ## Summary
    What does this PR do and why?

    ## Changes
    -

    ## How to test
    - From a fresh clone: steps to run

    ## Checklist
    - [ ] Runs from a fresh clone (README steps)
    - [ ] No secrets committed; `.env` only (and `.env.example` updated if needed)
    - [ ] Large artifacts tracked by LFS (`git lfs ls-files` shows expected files)
    - [ ] Clear, small diff; comments where useful
""")
open(".github/pull_request_template.md","w").write(tpl)
print("Wrote .github/pull_request_template.md")
```

(Optional) Require both teammates to review by setting **CODEOWNERS** (edit handles):

```
owners = """\
# Replace with your GitHub handles
* @teammate1 @teammate2
"""
open(".github/CODEOWNERS","w").write(owners)
print("Wrote .github/CODEOWNERS (edit handles!)")
```

Commit and push on a new branch (example: `chore/pr-template`), open a PR, and merge after review.

2.6.2 Part B — Add a large-file guard (simple Python script)

Create a small tool that **fails** if files > 10 MB are found **and** aren't tracked by LFS. This will be used manually for now (automation later in CI).

```python
# tools/guard_large_files.py
import os, subprocess, sys

LIMIT_MB = 10
ROOT = os.getcwd()

def lfs_tracked_paths():
    try:
        out = subprocess.check_output(["git", "lfs", "ls-files"], text=True)
        tracked = set()
        for line in out.strip().splitlines():
            # line format: "<oid> <path>"
            p = line.split(None, 1)[-1].strip()
            tracked.add(os.path.normpath(p))
        return tracked
    except Exception:
        return set()

def humanize(bytes_):
    return f"{bytes_/(1024*1024):.2f} MB"

lfs_set = lfs_tracked_paths()
bad = []
for dirpath, dirnames, filenames in os.walk(ROOT):
    # skip .git directory
    if ".git" in dirpath.split(os.sep):
```

```
        continue
    for fn in filenames:
        path = os.path.normpath(os.path.join(dirpath, fn))
        try:
            size = os.path.getsize(path)
        except FileNotFoundError:
            continue
        if size >= LIMIT_MB * 1024 * 1024:
            rel = os.path.relpath(path, ROOT)
            if rel not in lfs_set:
                bad.append((rel, size))

if bad:
    print("ERROR: Large non-LFS files found:")
    for rel, size in bad:
        print(f" - {rel} ({humanize(size)})")
    sys.exit(1)
else:
    print("OK: No large non-LFS files detected.")
```

Add a Makefile target to run it:

```
# Create/append Makefile target
from pathlib import Path
text = "\n\nguard:\n\tpython tools/guard_large_files.py\n"
p = Path("Makefile")
p.write_text(p.read_text() + text if p.exists() else text)
print("Added 'guard' target to Makefile")
```

Run locally/Colab:

```
!python tools/guard_large_files.py
```

Commit on a new branch (e.g., `chore/large-file-guard`), push, open PR, and merge after review.

2.6.3 Part C — Branch/PR practice (each student)

1. Each student creates **their own** branch (e.g., `docs/readme-username`) and:

- Adds a **“Development workflow”** section in `README.md` (1–2 paragraphs): how to clone, mount Drive in Colab, install requirements, and where outputs go.
- Adds themselves to `README.md` “Contributors” section with a GitHub handle link.

2. Push branch and open a PR.
3. Partner reviews the PR:

   - Leave at least **2 useful comments** (nits vs blockers).
   - Approve when ready; the author merges.

Expected files touched: `README.md`, `.github/pull_request_template.md`, optional `.github/CODEOWNERS`, `tools/guard_large_files.py`, `Makefile`.

2.6.4 Part D — Prove LFS is working

- On `main`, run:

```
!git lfs ls-files
```

- You should see `data/processed/sample_returns.parquet` (and any other tracked binaries).
- In the GitHub web UI, click the file to confirm it’s an **LFS pointer**, not full binary contents.

2.6.5 Submission checklist (pass/revise)

- Two merged PRs (template + guard) with clear titles and descriptions.
- README updated with development workflow and contributors.
- `git lfs ls-files` shows expected files.
- `tools/guard_large_files.py` present and passes (`OK`) on `main`.

2.7 Instructor checklist (before class)

- Ensure students have or can create a GitHub repo and add collaborators.
- Validate the lab sequence once in a fresh Colab runtime.
- Have example screenshots of: PR diff, LFS pointer file, successful `git lfs ls-files`.

2.8 Emphasize while teaching

- **Small PRs win.** Short diffs → fast, focused reviews.
- **Don't commit secrets.** `.env` only; keep `.env.example` up to date.
- **Use LFS sparingly** and purposefully—prefer regenerating big raw data.
- **Colab pushes:** use a **short-lived token**, and clear outputs after use.

Next session: **Quarto reporting** polish and pipeline hooks; soon after, **Unix automation** so `make get-data` can reproducibly fetch raw data for the unified-stocks project.

3 Session 3 — Quarto Reports (Python)

Below is a complete lecture package for **Session 3 — Quarto Reports (Python)** (75 minutes). It includes: timed agenda, key talking points, an **in-class lab with copy-paste code cells (Colab-friendly)**, and **homework with copy-paste code**. This session produces a **parameterized EDA report** for multiple tickers and publishes it to **GitHub Pages**.

> Assumptions: Students already have (from Sessions 1–2) a repo like `unified-stocks-teamX` in Drive (or they can create it now) and basic Git push workflow with a short-lived token. Today focuses on Quarto.

3.1 Session 3 — Quarto Reports (Python) — 75 minutes

3.1.1 Learning goals

By the end of class, students can:

1. Create a **parameterized** Quarto report (`.qmd`) that runs Python code.
2. Render a report from **Colab** using the **Quarto CLI** (with caching).
3. Pass parameters on the command line to re-render for different tickers/date ranges.
4. Configure a minimal Quarto **website** that builds to `docs/` and publish it via **GitHub Pages**.

3.2 Agenda (75 min)

- **(8 min)** Why Quarto for DS: literate programming, parameters, caching, publishing
- **(12 min)** Anatomy of a `.qmd`: YAML front matter, `params:`, code chunks, `execute:` options, figures
- **(35 min) In-class lab**: install Quarto in Colab → create `_quarto.yml` → write `reports/eda.qmd` → render for AAPL/MSFT → output to `docs/`

- **(10 min)** GitHub Pages walkthrough + troubleshooting + homework briefing
- **(10 min)** Buffer for hiccups (first Quarto install/render often needs a minute)

3.3 Talking points (for your slides)

Why Quarto

- One source of truth for code + prose + figures → reproducibility and explainability.
- Parameterization = fast re-runs with different inputs (ticker/horizon).
- Publishing to GitHub Pages gives a permanent, shareable artifact.

Key concepts

- **Front matter**:
  - `format:` controls HTML/PDF/RevealJS (we'll use HTML).
  - `execute:` controls caching, echo, warnings.
  - `params:` defines inputs; accessed as `params` dict in Python cells.
- **Performance**: enable `execute.cache: true` to avoid refetching/recomputing.
- **Publishing**: write to `docs/` then enable GitHub Pages (Settings → Pages → "Deploy from a branch" → `main` / `/docs`).

Ethics/footnote

- Financial data EDA here is **educational** only; not trading advice.

3.4 In-class lab (35 min)

> **Instructor tip**: Ask students to follow step-by-step. If they didn't complete Session 2's clone, they can create a fresh folder under Drive and initialize a new GitHub repo afterward.

3.4.1 0) Mount Drive and set repo paths

Run each block as a separate Colab cell.

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"  # <- change
REPO_NAME  = "unified-stocks-teamX"         # <- change
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"
REPO_URL   = f"https://github.com/{REPO_OWNER}/{REPO_NAME}.git"

import pathlib, os, subprocess
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)

if not pathlib.Path(REPO_DIR).exists():
    !git clone {REPO_URL} {REPO_DIR}
else:
    %cd {REPO_DIR}
    !git pull --ff-only
%cd {REPO_DIR}
```

3.4.2 1) Install Quarto CLI on Colab and verify

```
# Install Quarto CLI (one-time per Colab runtime)
!wget -q https://quarto.org/download/latest/quarto-linux-amd64.deb -O /tmp/quarto.deb
!dpkg -i /tmp/quarto.deb || apt-get -y -f install >/dev/null && dpkg -i /tmp/quarto.deb
!quarto --version
```

3.4.3 2) Minimal project config: `_quarto.yml` (website to `docs/`)

```
from textwrap import dedent
qproj = dedent("""\
project:
  type: website
  output-dir: docs

website:
```

```
  title: "Unified Stocks - EDA"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - href: reports/eda.qmd
        text: EDA (parametrized)

format:
  html:
    theme: cosmo
    toc: true
    code-fold: false

execute:
  echo: true
  warning: false
  cache: true
""")
open("_quarto.yml","w").write(qproj)
print(open("_quarto.yml").read())
```

Create a simple homepage:

```
index = """\
---
title: "Unified Stocks Project"
---

Welcome! Use the navigation to view the EDA report.

- **Stock set**: see `tickers_25.csv`
- **Note**: Educational use only - no trading advice.
"""
open("index.qmd","w").write(index)
print(open("index.qmd").read())
```

3.4.4 3) Create the parameterized EDA report: `reports/eda.qmd`

```python
import os, pathlib
pathlib.Path("reports/figs").mkdir(parents=True, exist_ok=True)

eda_qmd = """\
---
title: "Stock EDA"
format:
  html:
    toc: true
    number-sections: false
execute:
  echo: true
  warning: false
  cache: true
params:
  symbol: "AAPL"
  start_date: "2018-01-01"
  end_date: ""
  rolling: 20
---

::: callout-note
This report is parameterized. To change inputs without editing code, pass
`-P symbol:MSFT -P start_date:2019-01-01 -P end_date:2025-08-01 -P rolling:30` to `quarto ren
:::

## Setup



## Price over time



## Daily log returns - histogram



## Rolling mean & volatility (window = {params.rolling})



```

```
## Summary table



> **Note**: Educational use only. This is not trading advice.
> """
> open("reports/eda.qmd","w").write(eda\_qmd)
> print("Wrote reports/eda.qmd")
```

3.4.5 4) Render the report for one ticker (AAPL) and put outputs in `docs/`

```
# Single render with defaults (AAPL)
!quarto render reports/eda.qmd --output-dir docs/
```

Open the produced HTML (Colab file browser → `docs/reports/eda.html`). If the HTML is under `docs/reports/eda.html`, that's expected (Quarto keeps layout mirroring source folders).

3.4.6 5) Render for multiple tickers by passing parameters

```
# Render for MSFT with custom dates and rolling window
!quarto render reports/eda.qmd -P symbol:MSFT -P start_date:2019-01-01 -P end_date:2025-08-0

# Render for NVDA with a different window
!quarto render reports/eda.qmd -P symbol:NVDA -P start_date:2018-01-01 -P end_date:2025-08-0
```

This will create `docs/reports/eda.html` for the last render (Quarto overwrites the same output path by default). If you want **separate pages per ticker**, render to different filenames:

```
# Example: write MSFT to docs/reports/eda-MSFT.html via project copy
import shutil, os
shutil.copy("reports/eda.qmd", "reports/eda-MSFT.qmd")
!quarto render reports/eda-MSFT.qmd -P symbol:MSFT -P start_date:2019-01-01 -P end_date:2025
```

3.4.7 6) Add nav links to specific ticker pages (optional)

```
# Append MSFT page to navbar
from ruamel.yaml import YAML
yaml = YAML()
cfg = yaml.load(open("_quarto.yml"))
cfg["website"]["navbar"]["left"].append({"href": "reports/eda-MSFT.qmd", "text": "MSFT EDA"}
with open("_quarto.yml","w") as f:
    yaml.dump(cfg, f)
!quarto render --output-dir docs/
```

3.4.8 7) Commit and push site to GitHub (so Pages can serve `docs/`)

```
!git add _quarto.yml index.qmd reports/eda*.qmd reports/figs docs
!git status
!git commit -m "feat: add parameterized Quarto EDA and publish to docs/"
```

```
# Push using a short-lived fine-grained token (as in Session 2)
from getpass import getpass
token = getpass("GitHub token (not stored): ")
push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
!git push {push_url} HEAD:main
del token
```

3.4.9 8) Enable GitHub Pages (one-time, UI)

- On GitHub: **Settings → Pages**
  - Source: **Deploy from a branch**
  - Branch: `main`
  - Folder: `/docs`
- Save. Wait ~1–3 minutes. Your site will be live at the URL GitHub shows (usually `https://<owner>.github.io/<repo>/`).

3.5 Wrap-up (10 min)

- Re-rendering with `-P` lets you build many variants quickly.
- Keep **data fetches cached** and/or saved to files to speed up renders.
- Your team can add more pages (e.g., *Methodology*, *Results*, *Model Card*) and link them via `_quarto.yml`.

3.6 Homework (due before Session 4)

Goal: Enhance the EDA report with two features and publish distinct pages for **three** tickers from `tickers_25.csv`.

3.6.1 Part A — Add drawdown & simple regime shading

1. Edit `reports/eda.qmd`. After computing `df["log_return"]`, compute:

   - `cum_return` and **drawdown**
   - A simple **volatility regime** indicator (e.g., rolling std quantiles)

```python
# Add to the "Tidy & features" section in eda.qmd
df["cum_return"] = df["log_return"].cumsum().fillna(0.0)
peak = df["cum_return"].cummax()
df["drawdown"] = df["cum_return"] - peak

# Regime via rolling volatility terciles
vol = df["log_return"].rolling(ROLL, min_periods=ROLL//2).std()
q1, q2 = vol.quantile([0.33, 0.66])
def regime(v):
    if np.isnan(v): return "mid"
    return "low" if v < q1 else ("high" if v > q2 else "mid")
df["regime"] = [regime(v) for v in vol]
df["regime"].value_counts().to_frame("days").T
```

2. Add a **drawdown plot** and shade high-volatility regimes:

```python
# Drawdown plot
fig, ax = plt.subplots(figsize=(8,3))
ax.plot(df.index, df["drawdown"])
ax.set_title(f"{SYMBOL} - Drawdown (log-return cumulative)")
ax.set_xlabel("Date"); ax.set_ylabel("drawdown")
fig.tight_layout()
figpath = Path("reports/figs")/f"{SYMBOL}_drawdown.png"
fig.savefig(figpath, dpi=144)
figpath
```

```python
# Price with regime shading (simple)
fig, ax = plt.subplots(figsize=(8,3))
ax.plot(df.index, df["close"])
ax.set_title(f"{SYMBOL} - Price with High-Volatility Shading")
ax.set_xlabel("Date"); ax.set_ylabel("Price")

# Shade where regime == 'high'
mask = (df["regime"] == "high")
# merge contiguous regions
in_region = False
start = None
for i, (ts, is_high) in enumerate(zip(df.index, mask)):
    if is_high and not in_region:
        in_region = True
        start = ts
    if in_region and (not is_high or i == len(df)-1):
        end = df.index[i-1] if not is_high else ts
        ax.axvspan(start, end, alpha=0.15)  # shaded band
        in_region = False
fig.tight_layout()
figpath = Path("reports/figs")/f"{SYMBOL}_price_regimes.png"
fig.savefig(figpath, dpi=144)
figpath
```

3.6.2 Part B — Render three separate pages and link them in the navbar

1. Make copies of the report source so each produces its own page:

```python
import shutil
shutil.copy("reports/eda.qmd", "reports/eda-AAPL.qmd")
```

```
shutil.copy("reports/eda.qmd", "reports/eda-MSFT.qmd")
shutil.copy("reports/eda.qmd", "reports/eda-NVDA.qmd")
```

2. Render each with different parameters:

```
!quarto render reports/eda-AAPL.qmd -P symbol:AAPL -P start_date:2018-01-01 -P end_date:2025-
!quarto render reports/eda-MSFT.qmd -P symbol:MSFT -P start_date:2018-01-01 -P end_date:2025-
!quarto render reports/eda-NVDA.qmd -P symbol:NVDA -P start_date:2018-01-01 -P end_date:2025-
```

3. Add to the navbar in `_quarto.yml` and rebuild site:

```
from ruamel.yaml import YAML
yaml = YAML()
cfg = yaml.load(open("_quarto.yml"))
cfg["website"]["navbar"]["left"].extend([
  {"href": "reports/eda-AAPL.qmd", "text": "AAPL"},
  {"href": "reports/eda-MSFT.qmd", "text": "MSFT"},
  {"href": "reports/eda-NVDA.qmd", "text": "NVDA"},
])
with open("_quarto.yml","w") as f:
    yaml.dump(cfg, f)
!quarto render --output-dir docs/
```

4. Commit & push (use your short-lived token as before):

```
!git add reports/eda-*.qmd reports/figs _quarto.yml docs
!git commit -m "feat: EDA enhancements (drawdown/regimes) and pages for AAPL/MSFT/NVDA"
```

```
from getpass import getpass
token = getpass("GitHub token (not stored): ")
push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
!git push {push_url} HEAD:main
del token
```

5. Verify **GitHub Pages** shows navbar links and pages load.

3.6.3 Part C — Makefile convenience targets

Append these to your project `Makefile`:

```
report:
\tquarto render reports/eda.qmd --output-dir docs/

reports-trio:
\tquarto render reports/eda-AAPL.qmd -P symbol:AAPL -P start_date:2018-01-01 -P end_date:2025
\tquarto render reports/eda-MSFT.qmd -P symbol:MSFT -P start_date:2018-01-01 -P end_date:2025
\tquarto render reports/eda-NVDA.qmd -P symbol:NVDA -P start_date:2018-01-01 -P end_date:2025
```

On Colab, running `make` requires `make` to be available (it is). Otherwise, keep using `quarto render` commands.

3.6.4 Grading (pass/revise)

- `reports/eda.qmd` renders with parameters and caching enabled.
- At least **three** ticker pages rendered and linked in navbar.
- Drawdown and simple regime shading working on the EDA page(s).
- Site published via GitHub Pages (`docs/` present on `main` and live).

3.7 Instructor checklist (before class)

- Test the Quarto install/render flow once in a fresh Colab runtime.
- Have a screenshot of: `_quarto.yml`, rendered `docs/` tree, GitHub Pages settings.
- Remind students: if `yfinance` rate-limits, re-run or wait; the synthetic fallback ensures the page renders.

3.8 Emphasize while teaching

- **Parameters** make reports reusable; don't copy-paste notebooks for each ticker.
- **Cache** for speed; **`docs/`** for Pages.
- Keep figures saved under `reports/figs/` and referenced in the report.
- Keep secrets out of the repo; EDA uses public data only.

Next time (Session 4): a quick **RStudio Quarto cameo** and more **report hygiene** (citations, figure captions, alt text), then into **Unix automation**.

4 Session 4 — RStudio Quarto cameo + Report Hygiene

Below is a complete lecture package for **Session 4 — RStudio Quarto cameo + Report Hygiene** (75 minutes). It includes: a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. The goal is to make your Quarto site **clean, citable, accessible, and reproducible**—and to show (briefly) that **RStudio can render your Python-based Quarto project**.

> **Assumptions:**
>
> - Students already have a repo (e.g., `unified-stocks-teamX`) with the Quarto site scaffolding from Sessions 2–3.
> - Python-first course; the **RStudio cameo** demonstrates that Quarto is editor-agnostic (no R coding required).

4.1 Session 4 — RStudio cameo + Report Hygiene (75 min)

4.1.1 Learning goals

By the end of class, students can:

1. Render a **Python-only** Quarto report from **RStudio** (or RStudio Cloud) as a proof that Quarto is editor-agnostic.
2. Add **hygiene features** to the project: citations (`references.bib`), figure/table **captions + cross-references**, **alt text**, better site navigation, custom CSS, and **freeze/caching** for reproducibility.
3. Produce a **Data Dictionary** section that documents columns and dtypes, and reference it from the EDA page.
4. Render & publish the cleaned site to **GitHub Pages**.

4.2 Agenda (75 min)

- **(10 min)** Why report hygiene matters (credibility, accessibility, reusability)
- **(15 min) RStudio cameo**: Render the Python-based Quarto report in RStudio
- **(30 min) In-class lab** (Colab): add citations, cross-refs, alt text, freeze/caching, CSS, data dictionary, rebuild site
- **(10 min)** Wrap-up + troubleshooting + homework briefing
- **(10 min)** Buffer (for first-time installs or Git pushes)

4.3 Slides / talking points

4.3.1 Why hygiene?

- **Credibility:** citations + model/report lineage
- **Accessibility:** alt text, readable fonts, color-safe figures
- **Reusability:** parameters, freeze/caching, stable page links
- **Assessability:** clear captions, labeled figures & tables, cross-references

4.3.2 Quarto features we'll use

- **Captions & labels**: `#| label: fig-price`, `#| fig-cap: "Price over time"` → reference in text with `@fig-price`
- **Tables**: `#| label: tbl-summary`, `#| tbl-cap: "Summary statistics"` → reference with `@tbl-summary`
- **Alt text**: `#| fig-alt: "One-sentence description of the figure"`
- **Citations**: add `bibliography: references.bib` and cite with `[@key]`
- **Freeze**: project-level `freeze: auto` for deterministic rebuilds
- **Cache**: `execute: cache: true` to avoid redoing expensive steps
- **CSS**: small tweaks to readability (font size, code block width)

4.3.3 RStudio cameo (no R required)

- RStudio integrates Quarto; the **Render** button runs `quarto render` under the hood.
- Your `.qmd` can be Python-only; RStudio is just the IDE.

4.4 RStudio cameo (15 min, live demo steps)

Do this on the projector. Students observe; they can try later on their machines or RStudio Cloud.

1. **Open RStudio** (Desktop or Cloud).
2. **File → Open Project** and select your repo folder (`unified-stocks-teamX`).
3. Confirm Quarto: **Help → About Quarto** (or run `quarto --version` in the RStudio terminal).
4. Open `reports/eda.qmd`. Click **Render** (or run `quarto render reports/eda.qmd`).
5. Show the generated HTML preview. Note: no R code, just Python chunks.
6. Mention that **RMarkdown** is the predecessor; **Quarto** unifies Python & R (and more). We use **Quarto**.

4.5 In-class lab (30 min, Colab-friendly)

We'll: ensure Quarto CLI is present, upgrade `_quarto.yml` (freeze, bibliography, CSS), add `references.bib`, **rewrite EDA with captions/labels/alt text**, generate a **Data Dictionary**, re-render, and push to GitHub.

4.5.1 0) Mount Drive, set repo path, and ensure Quarto CLI

```python
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"  # <- change
REPO_NAME  = "unified-stocks-teamX"         # <- change
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"
REPO_URL   = f"https://github.com/{REPO_OWNER}/{REPO_NAME}.git"

import pathlib, os
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)

if not pathlib.Path(REPO_DIR).exists():
    !git clone {REPO_URL} {REPO_DIR}
%cd {REPO_DIR}
```

```
# Ensure Quarto CLI
!quarto --version || (wget -q https://quarto.org/download/latest/quarto-linux-amd64.deb -O /t
!quarto --version
```

4.5.2 1) Upgrade `_quarto.yml`: freeze, bibliography, CSS, nav polish

```
# Install ruamel.yaml for safe YAML edits
!pip -q install ruamel.yaml

from ruamel.yaml import YAML
from pathlib import Path

yaml = YAML()
cfg_path = Path("_quarto.yml")
if cfg_path.exists():
    cfg = yaml.load(cfg_path.read_text())
else:
    cfg = {"project": {"type": "website", "output-dir": "docs"},
           "website": {"title": "Unified Stocks", "navbar": {"left": [{"href":"index.qmd","te
           "format":{"html":{"theme":"cosmo","toc":True}}}

# Add/ensure features
cfg.setdefault("format", {}).setdefault("html", {})
cfg["format"]["html"]["toc"] = True
cfg["format"]["html"]["code-fold"] = False
cfg["format"]["html"]["toc-depth"] = 2
cfg["format"]["html"]["page-navigation"] = True
cfg["format"]["html"]["code-tools"] = True
cfg["format"]["html"]["fig-cap-location"] = "bottom"
cfg["format"]["html"]["tbl-cap-location"] = "top"
cfg["format"]["html"]["css"] = "docs/style.css"

cfg.setdefault("execute", {})
cfg["execute"]["echo"] = True
cfg["execute"]["warning"] = False
cfg["execute"]["cache"] = True

# Freeze: deterministic rebuilds until the source changes
cfg["project"]["freeze"] = "auto"
```

```
# Bibliography
cfg["bibliography"] = "references.bib"

# Ensure navbar has EDA link
nav = cfg.setdefault("website", {}).setdefault("navbar", {}).setdefault("left", [])
if not any(item.get("href") == "reports/eda.qmd" for item in nav if isinstance(item, dict)):
    nav.append({"href": "reports/eda.qmd", "text": "EDA"})

yaml.dump(cfg, open("_quarto.yml","w"))
print(open("_quarto.yml").read())
```

4.5.3 2) Add `references.bib` (sample entries; students will refine later)

```
refs = r"""@book{hyndman-fpp3,
  title = {Forecasting: Principles and Practice},
  author = {Hyndman, Rob J. and Athanasopoulos, George},
  edition = {3},
  year = {2021},
  url = {https://otexts.com/fpp3/}
}
@misc{quarto-docs,
  title = {Quarto Documentation},
  author = {{Posit}},
  year = {2025},
  url = {https://quarto.org/}
}
@misc{yfinance,
  title = {yfinance: Yahoo! Finance market data downloader},
  author = {Ran Aroussi},
  year = {2024},
  url = {https://github.com/ranaroussi/yfinance}
}
"""
open("references.bib","w").write(refs)
print(open("references.bib").read())
```

4.5.4 3) Overwrite `reports/eda.qmd` with captions, labels, alt text, citations, and cross-refs

This replaces the earlier EDA with a hygienic version. Feel free to adjust wording later.

``````python
from textwrap import dedent
eda = dedent("""\
---
title: "Stock EDA"
format:
  html:
    toc: true
    number-sections: false
execute:
  echo: true
  warning: false
  cache: true
params:
  symbol: "AAPL"
  start_date: "2018-01-01"
  end_date: ""
  rolling: 20
---

> *Educational use only - not trading advice.* Data pulled via **yfinance** [@yfinance].

This page is **parameterized**; see the **Parameters** section for usage.

## Setup

::: {.cell execution_count=1}
````` {.python .cell-code}
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import yfinance as yf
from pathlib import Path

SYMBOL = params.get("symbol", "AAPL")
START  = params.get("start_date", "2018-01-01")
END    = params.get("end_date", "")
``````
``````

```
ROLL   = int(params.get("rolling", 20))
if not END:
  END = pd.Timestamp.today().strftime("%Y-%m-%d")
```

4.6 Download and tidy

```
#| echo: true
try:
  data = yf.download(SYMBOL, start=START, end=END, auto_adjust=True, progress=False)
except Exception as e:
  # Synthetic fallback
  idx = pd.bdate_range(START, END)
  rng = np.random.default_rng(42)
  ret = rng.normal(0, 0.01, len(idx))
  price = 100 * np.exp(np.cumsum(ret))
  vol = rng.integers(1e5, 5e6, len(idx))
  data = pd.DataFrame({"Close": price, "Volume": vol}, index=idx)

df = (data.rename(columns=str.lower)[["close","volume"]]
        .dropna()
        .assign(log_return=lambda d: np.log(d["close"]).diff()))
df["roll_mean"] = df["log_return"].rolling(ROLL, min_periods=ROLL//2).mean()
df["roll_vol"]  = df["log_return"].rolling(ROLL, min_periods=ROLL//2).std()
df = df.dropna()
```

:::

4.7 Price over time

As shown in **Figure ?@fig-price**, prices vary over time with changing volatility.

4.8 Return distribution

Figure ?@fig-hist shows the return distribution; many assets exhibit heavy tails [(**hyndman-fpp3?**), pp. 20–21].

4.9 Rolling statistics (window = {params.rolling})

4.10 Summary table

See **Table ?@tbl-summary** for overall statistics.

4.11 Data dictionary

4.12 Parameters

This page accepts parameters: `symbol`, `start_date`, `end_date`, and `rolling`. You can re-render with:

```
quarto render reports/eda.qmd \\
  -P symbol:MSFT -P start_date:2019-01-01 -P end_date:2025-08-01 -P rolling:30
```

4.13 References

" " ") open("reports/eda.qmd","w").write(eda) print("Wrote reports/eda.qmd with hygiene features.")

```
### 4) Add a minimal CSS for readability
```python
from pathlib import Path
Path("docs").mkdir(exist_ok=True)
css = """\
/* Increase base font and widen code blocks slightly */
body { font-size: 1.02rem; }
pre code { white-space: pre-wrap; }
img { max-width: 100%; height: auto; }
"""
open("docs/style.css","w").write(css)
print("Wrote docs/style.css")
```
```

4.13.1 5) Render site to `docs/` and preview

```
!quarto render --output-dir docs/
```

Open `docs/reports/eda.html` in the Colab file browser to preview. Confirm:

- Captions under figures, tables titled at top
- Cross-refs like "Figure 1"/"Table 1" clickable
- "References" section at bottom with your 2–3 entries

4.13.2 6) Commit and push (short-lived token method)

```
!git add _quarto.yml references.bib reports/eda.qmd docs/style.css docs/
!git commit -m "chore: report hygiene (captions, cross-refs, alt text, freeze, bibliography,
```

```
from getpass import getpass
token = getpass("GitHub token (not stored): ")
push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
!git push {push_url} HEAD:main
del token
```

4.14 Wrap-up (10 min)

- Your report now has **citations**, **captions**, **cross-refs**, **alt text**, and **frozen** outputs for stable rebuilds.
- RStudio can render the exact same Python-based `.qmd`. Teams can mix editors without friction.
- Next: Unix automation and Makefile targets to run reports end-to-end.

4.15 Homework (due before Session 5)

Goal: Extend hygiene and add one analytic section—**ACF plot**—with proper captions/labels/alt text/citations.

4.15.1 Part A — Add an ACF figure with cross-ref + alt text

Append this code chunk to `reports/eda.qmd` after the "Rolling statistics" section:

````
```python
from getpass import getpass
token = getpass("GitHub token (not stored): ")
push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
!git push {push_url} HEAD:main
del token
```

### Grading (pass/revise)

* EDA page includes **ACF figure** with caption, label, and alt text; cross-referenced in te
* **Monthly returns** table present with caption/label; referenced in text.
* **At least two** new, relevant citations included and rendered under References.
* `freeze` and `cache` enabled; site renders to `docs/` and loads on GitHub Pages.

---

## Instructor checklist (before class)

* Confirm Quarto CLI install on your demo environment.
* Ensure you can open an existing Python-only `.qmd` in RStudio and click **Render** success
* Have a GitHub Pages site ready to show before/after hygiene improvements.

## Emphasize while teaching

* **Accessibility** is part of professionalism: always write **alt text**, don't rely on colo
* **Citations** are not optional for serious work; treat the report like a short paper.
* **Freeze + cache** save time and prevent accidental drift.
* RStudio is a **comfortable alternative editor** for Quarto even in a Python-only workflow.

Next up (Session 5): **Unix for data work**-shell power tools and Make automation to glue eve



`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4ifQ== -->`{=html}
````

```
```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiYm9va
```

Session 5 - Unix/Shell Essentials for Data Work

```````{.quarto-title-block template='/Users/yiwang/Applications/quarto/share/projects/book/p
---
title: Session 5 - Unix/Shell Essentials for Data Work

---
```

Below is a complete lecture package for **Session 5 — Unix/Shell Essentials for Data Work** (75 minutes). It includes: timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code cells**, and **homework with copy-paste code**. The lab works entirely in **Google Colab** using **Bash** commands and your course repo in **Google Drive**.

> **Scope today:** Filesystem navigation, pipes/redirects, `grep/sed/awk`, `sort|uniq|wc|cut|tr|head|tail|` `find|xargs`, regex basics, and a small **shell QA script** for CSV health checks.

## 4.16 Session 5 — Unix for Data Work (75 min)

### 4.16.1 Learning goals

By the end of class, students can:

1. 
2. Navigate and manipulate files safely from the shell (relative vs absolute paths, quoting).
3. 
4. Use **pipes** and **redirection** to build composable mini-pipelines.
5. 
6. Filter and transform text/CSV data with **`grep`**, **`sed`**, **`awk`**, and friends.
7. 
8. Find files with **`find`** and operate on them with **`xargs`** / **`-exec`** safely.
9. 
```

10. Write a small, **defensive Bash script** (`set -euo pipefail`) that performs data QA checks and returns a **non-zero exit code on failure**.
11. 

## 4.17 Agenda (75 min)

- 
- **(8 min)** Why shell for data science; mental model of pipelines
- 
- **(12 min)** Core commands & patterns: pipes/redirects, quoting, regex, `grep`/`sed`/`awk`
- 
- **(35 min) In-class lab** (Colab): filesystem → CSV pipelines → find/xargs → QA shell script
- 
- **(10 min)** Wrap-up, troubleshooting, and homework briefing
- 
- **(10 min)** Buffer for slow installs / student issues
- 

## 4.18 Slides / talking points (put these bullets on your slides)

**Why shell?**

- 
- Fast iteration for **data plumbing** (ETL glue) and **repeatable** ops.
- 
- Works on any POSIX host (your laptop, Colab VM, servers).
- 
- Lets you **compose** small tools with pipes: `producer | filter | summarize > report.txt`.
- 

### Mental model

-
- **Stream text** through commands. Each command reads **STDIN**, writes **STDOUT**; `|` connects them.
-
- **Redirection**: `>` (truncate to file), `>>` (append), `<` (read from file), `2>` (errors).
-
- **Exit code**: `0` success; non-zero = error. Use `&&` (only if success) and `||` (if failure).
-

### Quoting

-
- `"double quotes"` expand variables and backslashes;
-
- `'single quotes'` are **literal** (best for regex/cut/sed patterns);
-
- Always **quote paths** that might contain spaces: `"$FILE"`.
-

### Regex quick guide

-
- `^` start, `$` end, `.` any char, `*` 0+, `+` 1+, `?` 0/1, `[A-Z]` class, `(foo|bar)` alt.
-
- Use **`grep -E`** (ERE) for `+` and `|`. Plain **`grep`** is basic (BRE).
-

### CSV caution

-
- Unix tools are **line-oriented**. They're fine for simple CSVs (no embedded commas/quotes).
-

- For tricky CSVs, prefer Python/pandas. Today's examples are **simple CSVs**.
-

## 4.19 In-class lab (35 min)

> **Instructor tip:** Have students run these *as separate Colab cells*. Cells labeled "Bash" use `%%bash`. Cells labeled "Python" are only used to **generate a small synthetic CSV** we can play with offline (no API keys needed).

### 4.19.1 0) Mount Drive, set repo paths, and cd into the repo

**Python (Colab)**

```
from google.colab import drive drive.mount('/content/drive', force_remount=True)  REPO_OWNER
```

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" pwd ls -la
```

### 4.19.2 1) Create a small synthetic prices CSV to work with (safe & reproducible)

**Python**

```
# Generates data/raw/prices.csv with columns: ticker,date,adj_close,volume,log_return import
```

### 4.19.3 2) Pipes & redirects warm-up

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # How many
```

### 4.19.4 3) `grep` filters (basic and extended) + regex

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # All rows
```

> **Note:** We anchor `^` to the start of the line so `grep` doesn't match commas later in the row.

### 4.19.5 4) `sed` transformations (search/replace; in-place edits)

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # Make a c
```

### 4.19.6 5) `awk` for CSV summarization

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # Compute r
```

**Explanation for students**

- 
- `NR>1` skips header.
- 
- `sum[$1]` and `n[$1]` are associative arrays keyed by **ticker**.
- 
- We sort numerically on column 2 (mean) with `-k2,2n` or `nr` for descending.
- 

### 4.19.7 6) `sort | uniq` deduping and `comm` to compare lists

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # Unique ti
```

### 4.19.8 7) `find` and `xargs` (safe, null-terminated)

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # Show all
```

> **Pattern:** prefer `-print0 | xargs -0` to safely handle spaces/newlines in filenames.

### 4.19.9 8) Build a defensive CSV QA script and run it

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" mkdir -p sc
```

**Run the QA script**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" scripts/qa_
```

> Intentionally break it (optional): open `data/raw/prices.csv`, blank out a value, and re-run to watch it fail with non-zero exit code.

## 4.20 Wrap-up (10 min)

- 
- Shell is about **composable building blocks**. Learn 15 commands deeply; combine them fluently.
- 
- Prefer **null-safe** `find … -print0 | xargs -0` patterns; always quote variables: `"$FILE"`.
- 
- For complex CSV logic, **fall back to Python**; but shell shines for **quick filters and QA**.
- 
- We'll hook these into **Make** next session so one command runs your whole pipeline.
- 

## 4.21 Homework (due before Session 6)

**Goal:** Practice and codify shell workflows into your project:
(1) a *data stats* pipeline, (2) a *per-ticker split* utility, (3) a Makefile target, and (4) a short **shell-only EDA** text report.

### 4.21.1 Part A — Data stats pipeline (one-liners saved to files)

**Bash (Colab)**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" mkdir -p rep
```

### 4.21.2 Part B — Split per-ticker CSVs into `data/interim/ticker=XYZ/` directories

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" mkdir -p da
```

### 4.21.3 Part C — Add Makefile targets for QA and per-ticker split

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  # Append o
```

**Run the targets**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" make qa make
```

### 4.21.4 Part D — Shell-only mini EDA report

Create `reports/mini_eda.txt` with **three sections**: counts, top moves, mean returns.

**Bash**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"  {   echo "
```

### 4.21.5 Part E — Commit & push your changes (use your short-lived token as in Session 2)

**Bash + Python (getpass)**

```
%%bash set -euo pipefail cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX" git add scri
```

```
from getpass import getpass import os, subprocess token = getpass("GitHub token (not stored)
```

### 4.21.6 Grading (pass/revise)

- 
- `scripts/qa_csv.sh` exists, is executable, and **fails** on malformed CSV, **passes** on clean CSV.
- 
- `reports/data_counts.txt`, `reports/top10_abs_moves.csv`, `reports/mean_return_by_ticker.csv`, and `reports/mini_eda.txt` generated.
- 
- `make qa` and `make split-by-ticker` run successfully.
- 
- Per-ticker CSVs created under `data/interim/ticker=XYZ/`.
- 

## 4.22 Instructor checklist (before class)

- 
- Test the lab in a fresh Colab runtime; ensure Drive path matches.
- 
- Prepare a one-slide "cheat sheet" of `grep/sed/awk` flags used today.
- 
- (Optional) Verify `gzip` is available (it is on Colab).
- 

## 4.23 Emphasize while teaching

- 
- **Quote** variables and paths. Prefer `-print0 | xargs -0` with `find`.
- 
- **Fail fast** in scripts (`set -euo pipefail`) and return **non-zero** exit codes for CI.
- 
- Shell is for **plumbing**—it complements, not replaces, Python.
- 

**Next session (6):** "Make/just, rsync, ssh/tmux (survey)" and we'll wire `make get-data` and `make report` into a reproducible one-command pipeline.

# 5 Session 6 — Make/Automation + rsync + ssh/tmux (survey)

Below is a complete lecture package for **Session 6 — Make/Automation + rsync + ssh/tmux (survey)** (75 minutes). It includes: a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. By the end, students will have a **one-command pipeline** (`make all`) to fetch data, build features, render the Quarto EDA, and back up artifacts—plus practical notes on `rsync`, `ssh`, and `tmux`.

> **Assumptions:** You're using the same Drive-mounted repo from prior sessions (e.g., `unified-stocks-teamX`). The course remains Python-first, Colab-first. No trading advice—this is educational.

---

## 5.1 Session 6 — Make/Automation + rsync + ssh/tmux (75 min)

### 5.1.1 Learning goals

Students will be able to:

1. Explain how **Make** turns scripts into a **reproducible pipeline** via **targets, dependencies,** and **incremental builds**.
2. Create and use a **Makefile** with helpful defaults, variables, and a `help` target.
3. Use **rsync** to back up project artifacts and understand `--delete` and exclude patterns.
4. Understand the **ssh** key flow and a **tmux** workflow for long-running jobs (survey).

---

## 5.2 Agenda (75 min)

- **(12 min)** Slides: Why automation? How Make models dependencies & incremental builds; best practices
- **(10 min)** Slides: rsync fundamentals; `ssh` keys & config; `tmux` workflow (survey)
- **(33 min) In-class lab** (Colab): create scripts → author Makefile → run `make all` → rsync backup
- **(10 min)** Wrap-up & troubleshooting
- **(10 min)** Buffer

---

## 5.3 Slides / talking points (drop these bullets into your deck)

### 5.3.1 Why Make for DS pipelines?

- Encodes your workflow as **targets** that depend on **files** or other targets.
- **Incremental**: only rebuilds what changed.
- Plays nicely with CI (`make all` from a clean clone).
- Stable across OSes; no new runtime to learn.

**Core syntax**

```
target: dependencies
<TAB> recipe commands
```

- Use variables: `PY := python`, `QUARTO := quarto`.
- Use **PHONY** for meta-targets that don't produce files.
- Prefer **deterministic** outputs: fixed seeds, pinned versions, stable paths.

### 5.3.2 rsync basics

- `rsync -avh SRC/ DST/` → syncs directory trees, preserving metadata.
- `--delete` makes DST exactly match SRC (removes files not in SRC).
- `--exclude` to skip folders (`--exclude 'raw/'`).
- Remote with SSH: `rsync -avz -e ssh SRC/ user@host:/path/`.

### 5.3.3 ssh keys & tmux (survey)

- **Keys**: `ssh-keygen -t ed25519 -C "you@school.edu"`; add the **public** key to servers/GitHub; keep private key private.
- `~/.ssh/config` holds named hosts; `ssh myhpc` uses that stanza.
- **tmux**: start `tmux new -s train`; detach (`Ctrl-b d`); list (`tmux ls`); reattach (`tmux attach -t train`). Keeps jobs alive on remote shells.

---

## 5.4 In-class lab (33 min)

We'll create **three tiny scripts** and a **Makefile** that ties them together:

- `scripts/get_prices.py` → `data/raw/prices.csv` (Yahoo via `yfinance`, with synthetic fallback)
- `scripts/build_features.py` → `data/processed/features.parquet`
- `scripts/backup.sh` → rsync your artifacts to `backups/<timestamp>/`
- `Makefile` → `make all` runs end-to-end; `make report` renders Quarto; `make backup` syncs artifacts.

> **Instructor tip:** Have everyone run each block as a separate Colab cell.

### 5.4.1 0) Mount Drive and set repo path

```python
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"  # <- change
REPO_NAME  = "unified-stocks-teamX"         # <- change
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import pathlib, os, subprocess
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)
if not pathlib.Path(REPO_DIR).exists():
    raise SystemExit("Repo not found in Drive. Clone it first (see Session 2/3).")
os.chdir(REPO_DIR)
print("Working dir:", os.getcwd())
```

### 5.4.2 1) Quick tool checks (Make, rsync, Quarto)

```
import subprocess, shutil
def check(cmd):
    try:
        out = subprocess.check_output(cmd, text=True)
        print(cmd[0], "OK")
    except Exception as e:
        print(cmd[0], "NOT FOUND")
check(["make", "--version"])
check(["rsync", "--version"])
check(["quarto", "--version"])
```

If Quarto is missing, re-run the installer from Session 3 before `make report`.

### 5.4.3 2) Script: `scripts/get_prices.py`

```
from pathlib import Path
Path("scripts").mkdir(exist_ok=True)

get_py = r"""#!/usr/bin/env python
import argparse, sys, time
from pathlib import Path
import pandas as pd, numpy as np

def fetch_yf(ticker, start, end):
    import yfinance as yf
    df = yf.download(ticker, start=start, end=end, auto_adjust=True, progress=False)
    if df is None or df.empty:
        raise RuntimeError("empty")
    df = df.rename(columns=str.lower)[["close","volume"]]
    df.index.name = "date"
    df = df.reset_index()
    df["ticker"] = ticker
    return df[["ticker","date","close","volume"]]

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--tickers", default="tickers_25.csv")
```

```
    ap.add_argument("--start", default="2020-01-01")
    ap.add_argument("--end", default="")
    ap.add_argument("--out", default="data/raw/prices.csv")
    args = ap.parse_args()

    out = Path(args.out)
    out.parent.mkdir(parents=True, exist_ok=True)
    tickers = pd.read_csv(args.tickers)["ticker"].dropna().unique().tolist()

    rows = []
    for t in tickers:
        try:
            df = fetch_yf(t, args.start, args.end or None)
        except Exception:
            # synthetic fallback
            idx = pd.bdate_range(args.start, args.end or pd.Timestamp.today().date())
            rng = np.random.default_rng(42 + hash(t)%1000)
            r = rng.normal(0, 0.01, len(idx))
            price = 100*np.exp(np.cumsum(r))
            vol = rng.integers(1e5, 5e6, len(idx))
            df = pd.DataFrame({"ticker": t, "date": idx, "close": price, "volume": vol})
        df["date"] = pd.to_datetime(df["date"]).dt.date
        df["adj_close"] = df["close"]
        df = df.drop(columns=["close"])
        df["log_return"] = np.log(df["adj_close"]).diff().fillna(0.0)
        rows.append(df)

    allp = pd.concat(rows, ignore_index=True)
    allp = allp[["ticker","date","adj_close","volume","log_return"]]
    allp.to_csv(out, index=False)
    print("Wrote", out, "rows:", len(allp))

if __name__ == "__main__":
    sys.exit(main())
"""
open("scripts/get_prices.py","w").write(get_py)
import os, stat
os.chmod("scripts/get_prices.py", os.stat("scripts/get_prices.py").st_mode | stat.S_IEXEC)
print("Created scripts/get_prices.py")
```

### 5.4.4 3) Script: `scripts/build_features.py`

```
feat_py = r"""#!/usr/bin/env python
import argparse
from pathlib import Path
import pandas as pd, numpy as np

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--input", default="data/raw/prices.csv")
    ap.add_argument("--out", default="data/processed/features.parquet")
    ap.add_argument("--roll", type=int, default=20)
    args = ap.parse_args()

    df = pd.read_csv(args.input, parse_dates=["date"])
    df = df.sort_values(["ticker","date"])
    # groupwise lags
    df["r_1d"] = df["log_return"]
    for k in (1,2,3):
        df[f"lag{k}"] = df.groupby("ticker")["r_1d"].shift(k)
    df["roll_mean"] = (df.groupby("ticker")["r_1d"]
                       .rolling(args.roll, min_periods=args.roll//2).mean()
                       .reset_index(level=0, drop=True))
    df["roll_std"]  = (df.groupby("ticker")["r_1d"]
                       .rolling(args.roll, min_periods=args.roll//2).std()
                       .reset_index(level=0, drop=True))
    out = Path(args.out)
    out.parent.mkdir(parents=True, exist_ok=True)
    # Save compactly
    df.to_parquet(out, index=False)
    print("Wrote", out, "rows:", len(df))

if __name__ == "__main__":
    main()
"""
open("scripts/build_features.py","w").write(feat_py)
import os, stat
os.chmod("scripts/build_features.py", os.stat("scripts/build_features.py").st_mode | stat.S_
print("Created scripts/build_features.py")
```

### 5.4.5 4) Script: `scripts/backup.sh` (rsync)

```
backup_sh = r"""#!/usr/bin/env bash
# Sync selected artifacts to backups/<timestamp> using rsync.
# Usage: scripts/backup.sh [DEST_ROOT]
set -euo pipefail
ROOT="${1:-backups}"
STAMP="$(date +%Y%m%d-%H%M%S)"
DEST="${ROOT}/run-${STAMP}"
mkdir -p "$DEST"

# What to back up (adjust as needed)
INCLUDE=("data/processed" "reports" "docs")

for src in "${INCLUDE[@]}"; do
  if [[ -d "$src" ]]; then
    echo "Syncing $src -> $DEST/$src"
    rsync -avh --delete --exclude 'raw/' --exclude 'interim/' "$src"/ "$DEST/$src"/
  fi
done

echo "Backup complete at $DEST"
"""
open("scripts/backup.sh","w").write(backup_sh)
import os, stat
os.chmod("scripts/backup.sh", os.stat("scripts/backup.sh").st_mode | stat.S_IEXEC)
print("Created scripts/backup.sh")
```

### 5.4.6 5) Makefile (robust, with variables and `help`)

```
makefile = r"""# Makefile - unified-stocks
SHELL := /bin/bash
.SHELLFLAGS := -eu -o pipefail -c

PY := python
QUARTO := quarto

START ?= 2020-01-01
END   ?= 2025-08-01
```

```
ROLL  ?= 30

DATA_RAW := data/raw/prices.csv
FEATS    := data/processed/features.parquet
REPORT   := docs/reports/eda.html

# Default target
.DEFAULT_GOAL := help

.PHONY: help all clean clobber qa report backup

help: ## Show help for each target
	@awk 'BEGIN {FS = ":.*##"; printf "Available targets:\n"} /^[a-zA-Z0-9_\-]+:.*##/ {print

all: $(DATA_RAW) $(FEATS) report backup ## Run the full pipeline and back up artifacts

$(DATA_RAW): scripts/get_prices.py tickers_25.csv
	$(PY) scripts/get_prices.py --tickers tickers_25.csv --start $(START) --end $(END) --out

$(FEATS): scripts/build_features.py $(DATA_RAW) scripts/qa_csv.sh
	# Basic QA first
	scripts/qa_csv.sh $(DATA_RAW)
	$(PY) scripts/build_features.py --input $(DATA_RAW) --out $(FEATS) --roll $(ROLL)

report: $(REPORT) ## Render Quarto EDA to docs/
$(REPORT): reports/eda.qmd _quarto.yml docs/style.css
	$(QUARTO) render reports/eda.qmd -P symbol:AAPL -P start_date=$(START) -P end_date=$(END)
	@test -f $(REPORT) || (echo "Report not generated." && exit 1)

backup: ## Rsync selected artifacts to backups/<timestamp>/
	./scripts/backup.sh

clean: ## Remove intermediate artifacts (safe)
	rm -rf data/interim
	rm -rf data/processed/*.parquet || true

clobber: clean ## Remove generated reports and backups (dangerous)
	rm -rf docs/reports || true
	rm -rf backups || true
"""
open("Makefile","w").write(makefile)
print(open("Makefile").read())
```

> Note: The Makefile expects `scripts/qa_csv.sh` from Session 5. If a student missed it, set `scripts/qa_csv.sh` to a no-op or remove that dependency temporarily.

### 5.4.7 6) Try the pipeline

```python
import subprocess, os, textwrap, sys
print(subprocess.check_output(["make", "help"], text=True))
```

```python
# Fetch raw, build features, render report, back up artifacts
import subprocess
print(subprocess.check_output(["make", "all"], text=True))
```

Confirm:

- `data/raw/prices.csv` exists
- `data/processed/features.parquet` exists
- `docs/reports/eda.html` renders
- `backups/run-<timestamp>/` contains synced folders

### 5.4.8 7) (Optional) `just` command-runner

> **Optional**: If `just` is available on your system, create a `justfile` that mirrors common Make targets. On Colab, installation may or may not be available; this is just for reference.

```bash
%%bash
set -e
if ! command -v just >/dev/null 2>&1; then
  echo "just not found; skipping optional step."
  exit 0
fi
cat > justfile << 'EOF'
# justfile - optional convenience recipes
set shell := ["bash", "-eu", "-o", "pipefail", "-c"]

start := "2020-01-01"
end   := "2025-08-01"
roll  := "30"
```

```
help:
\t@echo "Recipes: get-data, features, report, all, backup"

get-data:
\tpython scripts/get_prices.py --tickers tickers_25.csv --start {{start}} --end {{end}} --ou

features:
\tbash -lc 'scripts/qa_csv.sh data/raw/prices.csv'
\tpython scripts/build_features.py --input data/raw/prices.csv --out data/processed/features

report:
\tquarto render reports/eda.qmd -P symbol:AAPL -P start_date={{start}} -P end_date={{end}} -

all: get-data features report

backup:
\t./scripts/backup.sh
EOF
echo "Wrote justfile (optional)."
```

### 5.4.9 8) ssh & tmux quickstarts (survey, run locally, not in Colab)

**ssh key generation (local terminal):**

```
ssh-keygen -t ed25519 -C "you@school.edu"
# Press enter to accept default path (~/.ssh/id_ed25519), set a passphrase (recommended)
cat ~/.ssh/id_ed25519.pub   # copy this PUBLIC key where needed (GitHub/servers)
```

**SSH config (`~/.ssh/config`, local):**

```
Host github
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_ed25519
  AddKeysToAgent yes
  IdentitiesOnly yes

Host myhpc
  HostName login.hpc.university.edu
  User your_netid
  IdentityFile ~/.ssh/id_ed25519
```

**Test GitHub SSH (local):**

```
ssh -T git@github.com
```

**tmux essentials (remote or local):**

```
tmux new -s train              # start session "train"
# ... run your long job ...
# detach: press Ctrl-b then d
tmux ls                        # list sessions
tmux attach -t train           # reattach
tmux kill-session -t train     # end session
```

---

## 5.5 Wrap-up (10 min)

- **Make** codifies your pipeline; the file graph serves as your dependency DAG.
- **Incremental** builds save time: edit one script → only downstream targets rebuild.
- **rsync** is your friend for backups/snapshots; be deliberate with `--delete`.
- **ssh/tmux**: you don't need them in Colab, but you will on campus servers/HPC.

---

## 5.6 Homework (due before Session 7)

**Goal:** Extend your automation with a tiny baseline training & evaluation step and polish your Makefile.

### 5.6.1 Part A — Add a minimal baseline trainer

Create `scripts/train_baseline.py` that learns a **linear regression** on lagged returns (toy baseline) and writes metrics.

```
train_py = r"""#!/usr/bin/env python
import argparse, json
from pathlib import Path
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--features", default="data/processed/features.parquet")
    ap.add_argument("--out-metrics", default="reports/baseline_metrics.json")
    args = ap.parse_args()

    df = pd.read_parquet(args.features)
    # Train/test split by date (last 20% for test)
    df = df.dropna(subset=["lag1","lag2","lag3","r_1d"])
    n = len(df)
    split = int(n*0.8)
    Xtr = df[["lag1","lag2","lag3"]].iloc[:split].values
    ytr = df["r_1d"].iloc[:split].values
    Xte = df[["lag1","lag2","lag3"]].iloc[split:].values
    yte = df["r_1d"].iloc[split:].values

    model = LinearRegression().fit(Xtr, ytr)
    pred = model.predict(Xte)
    mae = float(mean_absolute_error(yte, pred))

    Path("reports").mkdir(exist_ok=True)
    with open(args.out_metrics, "w") as f:
        json.dump({"model":"linear(lag1,lag2,lag3)","test_mae":mae,"n_test":len(yte)}, f, inc
    print("Wrote", args.out_metrics, "MAE:", mae)

if __name__ == "__main__":
    main()
"""
open("scripts/train_baseline.py","w").write(train_py)
import os, stat
os.chmod("scripts/train_baseline.py", os.stat("scripts/train_baseline.py").st_mode | stat.S_I
print("Created scripts/train_baseline.py")
```

### 5.6.2 Part B — Extend your Makefile with `train` and `all`

Append these to your `Makefile`:

```
# --- add after FEATS definition, near other targets ---

TRAIN_METRICS := reports/baseline_metrics.json

.PHONY: train
train: $(TRAIN_METRICS) ## Train toy baseline and write metrics

$(TRAIN_METRICS): scripts/train_baseline.py $(FEATS)
	$(PY) scripts/train_baseline.py --features $(FEATS) --out-metrics $(TRAIN_METRICS)

# Update 'all' to include 'train'
# all: $(DATA_RAW) $(FEATS) report backup   # OLD
# Replace with:
# all: $(DATA_RAW) $(FEATS) report train backup
```

Run:

```bash
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
make train
cat reports/baseline_metrics.json
```

### 5.6.3 Part C — Add a `help` description to every target and verify `make help`

Ensure each target in your `Makefile` has a `##` comment. Run:

```bash
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
make help
```

### 5.6.4 Part D — (Optional) Track small models/metrics with Git-LFS

If you decide to save model artifacts (e.g., `models/baseline.pkl`), track them:

```
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
git lfs track "models/*.pkl"
git add .gitattributes
git commit -m "chore: track small model files via LFS"
```

*(You can extend `train_baseline.py` to save `models/baseline.pkl` using `joblib`.)*

### 5.6.5 Part E — Commit & push

```
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
git add scripts/*.py scripts/backup.sh Makefile reports/baseline_metrics.json
git status
git commit -m "feat: automated pipeline with Make (data->features->report->train) and rsync
```

```python
from getpass import getpass
import subprocess
token = getpass("GitHub token (not stored): ")
REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"
REPO_NAME  = "unified-stocks-teamX"
push_url = f"https://{token}@github.com/{REPO_OWNER}/{REPO_NAME}.git"
subprocess.run(["git", "push", push_url, "HEAD:main"], check=True)
del token
```

### 5.6.6 Grading (pass/revise)

- `make all` runs from a fresh clone (with minimal edits for tokens/Quarto install) and produces: `data/raw/prices.csv`, `data/processed/features.parquet`, `docs/reports/eda.html`, `reports/baseline_metrics.json`, and a `backups/run-*/` snapshot.
- `Makefile` has **helpful `help` output** and **variables** (`START`, `END`, `ROLL`).
- `scripts/backup.sh` uses `rsync -avh --delete` and excludes `raw/` & `interim/`.
- (Optional) LFS tracking updated for models.

---

## 5.7 Instructor checklist (before class)

- Test the full lab in a fresh Colab runtime (data fetch, feature build, report render, backup).
- Have one slide that visually shows the **dependency graph**: `prices.csv` → `features.parquet` → `report/train`.
- Prepare a one-pager cheat sheet for `rsync` flags and `tmux` keystrokes.

## 5.8 Emphasize while teaching

- **Make** is a thin layer over shell commands—it doesn't replace Python; it orchestrates it.
- Keep targets **idempotent**: running twice shouldn't break; only rebuild on changes.
- Use `rsync` with care: `--delete` is powerful—double-check DEST paths.
- **ssh/tmux**: you'll want this the first time you run a long model on a remote machine.

# 6 Session 7 — SQL I: SQLite Schemas & Joins

Below is a complete lecture package for **Session 7 — SQL I: SQLite Schemas & Joins** (75 minutes). It includes: a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. By the end, students will have a **SQLite database** with a clean schema, constraints, indexes, and several **SELECT/JOIN** queries they can reuse from Python.

> **Assumptions:** You're continuing in the same Drive-mounted repo (e.g., `unified-stocks-teamX`). You have `data/raw/prices.csv` from prior sessions. If not, the lab includes a fallback generator.

---

## 6.1 Session 7 — SQL I: SQLite Schemas & Joins (75 min)

### 6.1.1 Learning goals

By the end of class, students can:

1. Create a **SQLite database** with proper **tables, primary keys, constraints,** and **indexes**.
2. Load CSV data into SQLite safely (parameterized inserts or `pandas.to_sql`), avoiding duplicates.
3. Write **SELECT** queries with **WHERE/ORDER/LIMIT** and basic **JOINs**.
4. Use **parameterized queries from Python** to avoid SQL injection.
5. Build a small **SQL I/O helper** to streamline queries from Python.

---

## 6.2 Agenda (75 minutes)

- **(10 min)** Why relational databases for DS; SQLite types; PK/constraints; indexes
- **(10 min)** DDL overview (CREATE TABLE/INDEX); transactions; parameterized queries
- **(35 min) In-class lab** (Colab): create `prices.db` → load `prices` + `meta` → write joins
- **(10 min)** Wrap-up & homework briefing
- **(10 min)** Buffer

---

## 6.3 Slides / talking points (add to your deck)

**Why SQLite for DS**

- Single file DB → easy to version, ship, query; no server admin.
- Stronger **guarantees** than loose CSVs: types, **constraints**, **unique keys**, **foreign keys**.
- Fast **filters/joins** with **indexes**; JIT queries from Python, R, or CLI.

**SQLite types & constraints**

- SQLite uses **dynamic typing** but honors affinities: `INTEGER`, `REAL`, `TEXT`, `BLOB`.
- Use **PRIMARY KEY** (uniqueness + index), **NOT NULL**, and **CHECK** (e.g., volume 0).
- Turn on **foreign keys**: `PRAGMA foreign_keys = ON;`

**Indexes & performance**

- Index columns used in **joins** and **filters**.
- Composite PK `(ticker, date)` makes common lookups fast.

**What NOT to commit**

- Large `.db` files. Keep DB small or regenerate from CSV with a script.
- If you must version a small DB, ensure **Git-LFS** tracks `data/*.db` (we set this in Session 2).

---

## 6.4 In-class lab (35 min)

> **Instructor tip:** Have students run each block as its own Colab cell. Commands that start with `!` run in the Colab shell; the rest are Python.

### 6.4.1 0) Mount Drive & enter repo

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"   # <- change
REPO_NAME  = "unified-stocks-teamX"          # <- change
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, pandas as pd, numpy as np
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)
assert pathlib.Path(REPO_DIR).exists(), "Repo not found in Drive. Clone it first."
os.chdir(REPO_DIR)
print("Working dir:", os.getcwd())
```

### 6.4.2 1) Ensure prerequisites & create a small `prices.csv` if missing

```
# Ensure pandas and sqlite3 are available (sqlite3 is in stdlib)
import pandas as pd, sqlite3, numpy as np, os
from pathlib import Path

Path("data/raw").mkdir(parents=True, exist_ok=True)
if not Path("data/raw/prices.csv").exists():
    print("No prices.csv found; generating a small synthetic one.")
    tickers = ["AAPL","MSFT","NVDA","AMZN","GOOGL"]
    dates = pd.bdate_range("2022-01-03", periods=120)
    rng = np.random.default_rng(7)
    frames=[]
    for t in tickers:
        r = rng.normal(0, 0.01, len(dates))
        price = 100*np.exp(np.cumsum(r))
        vol = rng.integers(1e5, 5e6, len(dates))
```

```
        df = pd.DataFrame({"ticker": t, "date": dates, "adj_close": price, "volume": vol})
        df["log_return"] = np.log(df["adj_close"]).diff().fillna(0)
        frames.append(df)
    pd.concat(frames, ignore_index=True).to_csv("data/raw/prices.csv", index=False)

# Show a peek
pd.read_csv("data/raw/prices.csv").head()
```

### 6.4.3 2) Design schema & create the database `data/prices.db`

We'll use two tables:

- `meta(ticker TEXT PRIMARY KEY, name TEXT, sector TEXT NOT NULL)`
- `prices(ticker TEXT NOT NULL, date TEXT NOT NULL, adj_close REAL NOT NULL, volume INTEGER NOT NULL, log_return REAL NOT NULL, PRIMARY KEY (ticker,date), FOREIGN KEY (ticker) REFERENCES meta(ticker))`

```
import sqlite3, textwrap, os
from pathlib import Path

db_path = Path("data/prices.db")
if db_path.exists(): db_path.unlink()  # start fresh for class; remove this in real life
con = sqlite3.connect(db_path)
cur = con.cursor()

# Turn on foreign keys
cur.execute("PRAGMA foreign_keys = ON;")
# (Optional) WAL can help concurrency; not critical here
cur.execute("PRAGMA journal_mode = WAL;")

ddl = textwrap.dedent("""
CREATE TABLE meta (
  ticker TEXT PRIMARY KEY,
  name   TEXT,
  sector TEXT NOT NULL
);

CREATE TABLE prices (
  ticker     TEXT NOT NULL,
  date       TEXT NOT NULL,               -- ISO 'YYYY-MM-DD'
  adj_close  REAL NOT NULL CHECK (adj_close >= 0),
```

```
  volume     INTEGER NOT NULL CHECK (volume >= 0),
  log_return REAL NOT NULL,
  PRIMARY KEY (ticker, date),
  FOREIGN KEY (ticker) REFERENCES meta(ticker)
);

-- Index to speed up date-range scans across all tickers
CREATE INDEX IF NOT EXISTS idx_prices_date ON prices(date);
""")
cur.executescript(ddl)
con.commit()
print("Created:", db_path)
```

### 6.4.4 3) Populate `meta` (try yfinance sector; fallback to synthetic)

```python
import pandas as pd, numpy as np
import warnings
warnings.filterwarnings("ignore")

# Read tickers (from existing CSV or fallback)
if Path("tickers_25.csv").exists():
    tickers = pd.read_csv("tickers_25.csv")["ticker"].dropna().unique().tolist()
else:
    tickers = pd.read_csv("data/raw/prices.csv")["ticker"].dropna().unique().tolist()

def fetch_sector_map(tickers):
    try:
        import yfinance as yf
        out=[]
        for t in tickers:
            info = yf.Ticker(t).info or {}
            name  = info.get("shortName") or info.get("longName") or t
            sector= info.get("sector") or "Unknown"
            out.append({"ticker": t, "name": name, "sector": sector})
        return pd.DataFrame(out)
    except Exception:
        pass
    # Fallback: deterministic synthetic sectors
    sectors = ["Technology","Financials","Healthcare","Energy","Consumer"]
    rng = np.random.default_rng(42)
```

```
    return pd.DataFrame({
        "ticker": tickers,
        "name": tickers,
        "sector": [sectors[i % len(sectors)] for i in range(len(tickers))]
    })

meta_df = fetch_sector_map(tickers)
meta_df.head()
```

```
# Insert meta with parameterized query
with con:
    con.executemany(
        "INSERT INTO meta(ticker, name, sector) VALUES(?, ?, ?)",
        meta_df[["ticker","name","sector"]].itertuples(index=False, name=None)
    )
print(pd.read_sql_query("SELECT * FROM meta LIMIT 5;", con))
```

### 6.4.5 4) Load `data/raw/prices.csv` into a staging DataFrame and insert into `prices`

We'll use **parameterized bulk insert** (`executemany`) which is fast and safe.

```
prices = pd.read_csv("data/raw/prices.csv", parse_dates=["date"])
# Normalize date to ISO text
prices["date"] = prices["date"].dt.strftime("%Y-%m-%d")
# Keep only needed columns and ensure order matches table
prices = prices[["ticker","date","adj_close","volume","log_return"]]

# Optional: drop duplicates to respect PK before insert
prices = prices.drop_duplicates(subset=["ticker","date"]).reset_index(drop=True)
len(prices)
```

```
# Bulk insert inside one transaction; ignore rows violating FK or PK (e.g., duplicates)
with con:
    con.executemany(
        "INSERT OR IGNORE INTO prices(ticker,date,adj_close,volume,log_return) VALUES(?,?,?,?
        prices.itertuples(index=False, name=None)
    )

# Quick counts
```

```
print(pd.read_sql_query("SELECT COUNT(*) AS nrows FROM prices;", con))
print(pd.read_sql_query("SELECT ticker, COUNT(*) AS n FROM prices GROUP BY ticker ORDER BY n
```

### 6.4.6 5) Sanity queries (filters, order, limit)

```
q1 = """
SELECT ticker, date, adj_close, volume
FROM prices
WHERE ticker = ? AND date BETWEEN ? AND ?
ORDER BY date ASC
LIMIT 5;
"""
print(pd.read_sql_query(q1, con, params=["AAPL","2022-03-01","2022-06-30"]))
```

```
# Top 10 absolute daily moves for a chosen ticker
q2 = """
SELECT p.ticker, p.date, p.log_return, ABS(p.log_return) AS abs_move
FROM prices AS p
WHERE p.ticker = ?
ORDER BY abs_move DESC
LIMIT 10;
"""
print(pd.read_sql_query(q2, con, params=["NVDA"]))
```

### 6.4.7 6) JOIN with `meta` (per-sector summaries)

```
# Mean |std| of daily returns per sector over a date range
q3 = """
SELECT m.sector,
       AVG(ABS(p.log_return)) AS mean_abs_return,
       AVG(p.log_return)      AS mean_return,
       STDDEV(p.log_return)   AS std_return
FROM prices p
JOIN meta   m ON p.ticker = m.ticker
WHERE p.date BETWEEN ? AND ?
GROUP BY m.sector
ORDER BY mean_abs_return DESC;
"""
```

```
# SQLite doesn't have STDDEV by default; fallback using variance formula via window? We'll co
df = pd.read_sql_query("""
SELECT m.sector, p.log_return
FROM prices p JOIN meta m ON p.ticker = m.ticker
WHERE p.date BETWEEN ? AND ?;
""", con, params=["2022-01-01","2025-08-01"])
agg = (df.assign(abs=lambda d: d["log_return"].abs())
         .groupby("sector")
         .agg(mean_abs_return=("abs","mean"),
              mean_return=("log_return","mean"),
              std_return=("log_return","std"))
         .sort_values("mean_abs_return", ascending=False))
agg
```

### 6.4.8 7) Create a view for convenience & test uniqueness constraint

```
# View: latest available date per ticker
with con:
    con.execute("""
    CREATE VIEW IF NOT EXISTS latest_prices AS
    SELECT p.*
    FROM prices p
    JOIN (
      SELECT ticker, MAX(date) AS max_date
      FROM prices
      GROUP BY ticker
    ) t ON p.ticker = t.ticker AND p.date = t.max_date;
    """)
pd.read_sql_query("SELECT * FROM latest_prices ORDER BY ticker LIMIT 10;", con)
```

```
# Demonstrate the UNIQUE/PK constraint: inserting a duplicate row should be ignored or fail
import sqlite3
row = pd.read_sql_query("SELECT * FROM prices LIMIT 1;", con).iloc[0].to_dict()
try:
    with con:
        con.execute(
            "INSERT INTO prices(ticker,date,adj_close,volume,log_return) VALUES(?,?,?,?,?)",
            (row["ticker"], row["date"], row["adj_close"], row["volume"], row["log_return"])
        )
    print("Unexpected: duplicate insert succeeded (should not).")
```

```
except sqlite3.IntegrityError as e:
    print("IntegrityError as expected:", e)
```

### 6.4.9 8) A tiny SQL I/O helper for your project

```
from pathlib import Path
Path("src").mkdir(exist_ok=True)
Path("src/projectname").mkdir(parents=True, exist_ok=True)

sqlio_py = """\
from __future__ import annotations
import sqlite3
import pandas as pd
from contextlib import contextmanager
from pathlib import Path

DB_PATH = Path("data/prices.db")

@contextmanager
def connect(db_path: str | Path = DB_PATH):
    con = sqlite3.connect(str(db_path))
    con.execute("PRAGMA foreign_keys = ON;")
    try:
        yield con
    finally:
        con.close()

def query_df(sql: str, params: tuple | list | None = None, db_path: str | Path = DB_PATH) ->
    with connect(db_path) as con:
        return pd.read_sql_query(sql, con, params=params)

def sector_summary(start: str, end: str, db_path: str | Path = DB_PATH) -> pd.DataFrame:
    sql = '''
    SELECT m.sector, p.log_return
    FROM prices p JOIN meta m ON p.ticker = m.ticker
    WHERE p.date BETWEEN ? AND ?;
    '''
    df = query_df(sql, [start, end], db_path)
    if df.empty:
        return df
```

```
    g = df.assign(abs=lambda d: d["log_return"].abs()).groupby("sector")
    return g.agg(mean_abs_return=("abs","mean"),
                 mean_return=("log_return","mean"),
                 std_return=("log_return","std")).reset_index()
"""
open("src/projectname/sqlio.py","w").write(sqlio_py)
print("Wrote src/projectname/sqlio.py")
```

Quick test:

```
from src.projectname.sqlio import sector_summary
sector_summary("2022-01-01","2025-08-01").head()
```

> **Note on versioning:** If `data/prices.db` stays small (a few MB), you may commit it via **Git-LFS** (we tracked `data/*.db` in Session 2). Otherwise, **do not commit**—rebuild from CSV with a script (homework).

---

## 6.5 Wrap-up (10 min)

- You now have a **relational core** for the project.
- Use **PK + constraints** to prevent silent data corruption.
- Use **parameterized queries** from Python.
- Next session: **SQL II — Window functions & `pandas.read_sql` workflows** (rolling stats, LAG/LEAD).

---

## 6.6 Homework (due before Session 8)

**Goal:** Make database creation reproducible, add metadata, write joins you'll reuse later, and hook it into your Makefile.

### 6.6.1 Part A — Script to (re)build the DB

Create `scripts/build_db.py` that **creates tables** and **loads CSVs** deterministically.

```python
# scripts/build_db.py
#!/usr/bin/env python
import argparse, sys, textwrap, sqlite3
from pathlib import Path
import pandas as pd, numpy as np

DDL = textwrap.dedent("""
PRAGMA foreign_keys = ON;
CREATE TABLE IF NOT EXISTS meta (
  ticker TEXT PRIMARY KEY,
  name   TEXT,
  sector TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS prices (
  ticker     TEXT NOT NULL,
  date       TEXT NOT NULL,
  adj_close  REAL NOT NULL CHECK (adj_close >= 0),
  volume     INTEGER NOT NULL CHECK (volume >= 0),
  log_return REAL NOT NULL,
  PRIMARY KEY (ticker,date),
  FOREIGN KEY (ticker) REFERENCES meta(ticker)
);
CREATE INDEX IF NOT EXISTS idx_prices_date ON prices(date);
""")

def load_meta(con, tickers_csv: Path):
    if tickers_csv.exists():
        tks = pd.read_csv(tickers_csv)["ticker"].dropna().unique().tolist()
    else:
        raise SystemExit(f"tickers CSV not found: {tickers_csv}")
    sectors = ["Technology","Financials","Healthcare","Energy","Consumer"]
    meta = pd.DataFrame({
        "ticker": tks,
        "name": tks,
        "sector": [sectors[i % len(sectors)] for i in range(len(tks))]
    })
    with con:
        con.executemany("INSERT OR REPLACE INTO meta(ticker,name,sector) VALUES(?,?,?)",
                        meta.itertuples(index=False, name=None))
```

```
def load_prices(con, prices_csv: Path):
    df = pd.read_csv(prices_csv, parse_dates=["date"])
    df["date"] = df["date"].dt.strftime("%Y-%m-%d")
    df = df[["ticker","date","adj_close","volume","log_return"]].drop_duplicates(["ticker","d
    with con:
        con.executemany(
            "INSERT OR REPLACE INTO prices(ticker,date,adj_close,volume,log_return) VALUES(?
            df.itertuples(index=False, name=None)
        )

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--db", default="data/prices.db")
    ap.add_argument("--tickers", default="tickers_25.csv")
    ap.add_argument("--prices", default="data/raw/prices.csv")
    args = ap.parse_args()

    Path(args.db).parent.mkdir(parents=True, exist_ok=True)
    con = sqlite3.connect(args.db)
    con.executescript(DDL)
    load_meta(con, Path(args.tickers))
    load_prices(con, Path(args.prices))
    con.close()
    print("Built DB:", args.db)

if __name__ == "__main__":
    sys.exit(main())
```

Make it executable:

```
import os, stat, pathlib
p = pathlib.Path("scripts/build_db.py")
os.chmod(p, os.stat(p).st_mode | stat.S_IEXEC)
print("Ready:", p)
```

### 6.6.2 Part B — Add Makefile target `db` and a small SQL report

Append to your `Makefile`:

```
DB := data/prices.db

.PHONY: db sql-report
db: ## Build/refresh SQLite database from CSVs
\tpython scripts/build_db.py --db $(DB) --tickers tickers_25.csv --prices data/raw/prices.csv

sql-report: db ## Generate a simple SQL-driven CSV summary
\tpython - << 'PY'
import pandas as pd, sqlite3, os
con = sqlite3.connect("data/prices.db")
df = pd.read_sql_query(\"\"\"\nSELECT m.sector, COUNT(*) AS n_obs, AVG(ABS(p.log_return)) AS
os.makedirs("reports", exist_ok=True)
df.to_csv("reports/sql_sector_summary.csv", index=False)
print(df.head())
con.close()
PY
```

Run:

```
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
make db
make sql-report
```

### 6.6.3 Part C — Write 3 JOIN queries (save as `.sql` under `sql/`)

Create a folder `sql/` and add:

1. `sector_top_moves.sql`: top 10 absolute daily moves per sector (date, ticker, abs_move).
2. `ticker_activity.sql`: per-ticker counts, min/max date.
3. `range_summary.sql`: for a given date range (use placeholders), mean/std of returns by ticker and sector.

Example (1):

```
-- sql/sector_top_moves.sql
SELECT m.sector, p.ticker, p.date, p.log_return, ABS(p.log_return) AS abs_move
FROM prices p JOIN meta m ON p.ticker = m.ticker
ORDER BY abs_move DESC
LIMIT 10;
```

Then a small Python launcher to run any `.sql` file with optional parameters:

```
# scripts/run_sql.py
#!/usr/bin/env python
import argparse, sqlite3, pandas as pd
from pathlib import Path

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--db", default="data/prices.db")
    ap.add_argument("--sqlfile", required=True)
    ap.add_argument("--params", nargs="*", default=[])
    ap.add_argument("--out", default="")
    args = ap.parse_args()

    sql = Path(args.sqlfile).read_text()
    con = sqlite3.connect(args.db)
    df = pd.read_sql_query(sql, con, params=args.params or None)
    con.close()
    if args.out:
        Path(args.out).parent.mkdir(parents=True, exist_ok=True)
        df.to_csv(args.out, index=False)
    print(df.head())

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

Run a demo:

```
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
python scripts/run_sql.py --sqlfile sql/sector_top_moves.sql --out reports/sector_top_moves.
```

### 6.6.4 Part D — (Stretch) Create a calendar table & missing-day check

Create `calendar(date TEXT PRIMARY KEY)` covering min→max date in `prices`, and write a query that counts **missing business days** per ticker (join `calendar` LEFT JOIN `prices`). Save result to `reports/missing_days.csv`.

*Hint: build the calendar in Python with `pd.bdate_range()`; insert into `calendar`; then `SELECT c.date, p.ticker FROM calendar c LEFT JOIN prices p ... WHERE p.date IS NULL`.*

### 6.6.5 Part E — Commit & push (use the short-lived token flow from Session 2)

Recommended files to add:

- `scripts/build_db.py`, `scripts/run_sql.py`, `sql/*.sql`, updated `Makefile`, `reports/sql_sector_summary.csv`
- Optionally **do not** commit `data/prices.db` if large; if small and you must commit, ensure LFS is tracking `data/*.db`.

---

### 6.6.6 Grading (pass/revise)

- `data/prices.db` builds from `make db` and contains **meta** + **prices** with PK `(ticker,date)` and FK to `meta`.
- `reports/sql_sector_summary.csv` generated by `make sql-report`.
- `sql/sector_top_moves.sql`, `sql/ticker_activity.sql`, `sql/range_summary.sql` present and runnable via `scripts/run_sql.py`.
- If stretch completed: `calendar` table + `reports/missing_days.csv`.

---

## 6.7 Instructor checklist (before class)

- Run the entire lab once in a fresh Colab runtime with a small synthetic `prices.csv` to ensure speed.
- Prepare a one-slide schema diagram (`meta` `prices`) and a slide showing an index scan vs table scan (EXPLAIN output optional).
- Remind students: **don't commit big DBs**; keep `build_db.py` as the source of truth.

## 6.8 Emphasize while teaching

- **Schema first**: clean DDL prevents downstream headaches.
- **Constraints** are your guardrails; test them (we did with a duplicate insert).
- **Parameterize** queries; never string-concat user inputs into SQL.
- Keep SQLite for **analysis**; push heavy analytics to Python/Polars when needed.

Next time (Session 8): **SQL II — Window functions & `pandas.read_sql` workflows** (LAG/LEAD, rolling stats, and SQL pandas round-trips).

# 7 Session 8 — SQL II: Window Functions & `pandas.read_sql` Workflows

Below is a complete lecture package for **Session 8 — SQL II: Window Functions & `pandas.read_sql` Workflows** (75 minutes). It includes: a timed agenda, slides/talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. Today you'll compute **lags, leads, rolling stats, and top-k queries** in SQLite using **window functions**, then pull results into pandas for downstream use.

> **Assumptions:**
>
> - You're in the same Drive-mounted repo (e.g., `unified-stocks-teamX`).
> - You have `data/prices.db` from Session 7. If not, the lab includes a small fallback to build it from `data/raw/prices.csv`.
> - Educational use only — not trading advice.

---

## 7.1 Session 8 — SQL II: Window Functions & `pandas.read_sql` (75 min)

### 7.1.1 Learning goals

By the end of class, students can:

1. Explain and use **window functions**: `LAG`, `LEAD`, `ROW_NUMBER`, and aggregates with `OVER (PARTITION BY … ORDER BY … ROWS …)`.
2. Compute **rolling means/variance** and **multi-lag features** per ticker **without leakage**.
3. Use `WINDOW` named frames to avoid repetition and errors.
4. Run parameterized SQL from Python with `pandas.read_sql_query`, and optionally register a custom SQL function (e.g., `SQRT`).
5. Evaluate query performance basics with `EXPLAIN QUERY PLAN`.

---

## 7.2 Agenda (75 min)

- **(10 min)** Window functions: mental model & syntax (`PARTITION BY`, `ORDER BY`, `ROWS` frames)
- **(10 min)** Patterns for time series: lags, leads, rolling stats, top-k per group
- **(35 min) In-class lab** (Colab): lags/leads → rolling mean/variance → z-scores → top days per ticker → pull into pandas and save features
- **(10 min)** Wrap-up, performance notes (`EXPLAIN QUERY PLAN`), homework briefing
- **(10 min)** Buffer

---

## 7.3 Slides / talking points

### What's a window?

- A **window** lets an aggregate or analytic function see a **row + its neighbors** without collapsing rows.
- Template:

```
func(expr) OVER (
   PARTITION BY key
   ORDER BY time
   ROWS BETWEEN N PRECEDING AND CURRENT ROW
)
```

- **`PARTITION BY`** = per-group window; **`ORDER BY`** = sequence; **`ROWS`** = how many rows to include.

### Window vs GROUP BY

- `GROUP BY` returns **one row per group**; `OVER (…)` returns **one row per input row** with extra columns.

### Time-series patterns

- **Lags:** `LAG(x, k)` → previous `k` rows features at $t$ that use only info $t$.
- **Leads:** `LEAD(x, k)` → future `k` rows *labels* (don't leak into features).
- **Rolling stats:** `AVG(x) OVER (… ROWS BETWEEN w-1 PRECEDING AND CURRENT ROW)`; rolling variance via `AVG(x*x) - AVG(x)^2`.
- **Top-k per group:** compute `ROW_NUMBER() OVER (PARTITION BY key ORDER BY score DESC)` and filter `WHERE rn<=k`.

**ROWS vs RANGE**

- Use `ROWS` for fixed-length windows on ordered rows (what we need).
- **Time-based windows** (e.g., last 30 calendar days) require different techniques in SQLite (correlated subquery); we'll note but not use today.

---

## 7.4 In-class lab (35 min)

Run each block as its own Colab cell. Adjust `REPO_OWNER/REPO_NAME` first.

### 7.4.1 0) Mount Drive, open DB, and ensure it exists

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_OWNER = "YOUR_GITHUB_USERNAME_OR_ORG"     # <- change
REPO_NAME  = "unified-stocks-teamX"            # <- change
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, sqlite3, pandas as pd, numpy as np, math, textwrap
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)
assert pathlib.Path(REPO_DIR).exists(), "Repo not found; clone it first."
os.chdir(REPO_DIR)
print("Working dir:", os.getcwd())

# Ensure DB exists (fallback: build from CSV)
db_path = pathlib.Path("data/prices.db")
if not db_path.exists():
    print("prices.db not found; attempting minimal build from data/raw/prices.csv …")
    pathlib.Path("data").mkdir(exist_ok=True)
    con = sqlite3.connect(db_path)
    con.execute("PRAGMA foreign_keys = ON;")
    con.executescript("""
    CREATE TABLE IF NOT EXISTS meta (
      ticker TEXT PRIMARY KEY,
      name   TEXT,
```

```
  sector TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS prices (
  ticker     TEXT NOT NULL,
  date       TEXT NOT NULL,
  adj_close  REAL NOT NULL CHECK (adj_close >= 0),
  volume     INTEGER NOT NULL CHECK (volume >= 0),
  log_return REAL NOT NULL,
  PRIMARY KEY (ticker,date),
  FOREIGN KEY (ticker) REFERENCES meta(ticker)
);
CREATE INDEX IF NOT EXISTS idx_prices_date ON prices(date);
""")
# Minimal meta from tickers_25 or from CSV
if pathlib.Path("tickers_25.csv").exists():
    tks = pd.read_csv("tickers_25.csv")["ticker"].dropna().unique().tolist()
else:
    raw = pd.read_csv("data/raw/prices.csv")
    tks = raw["ticker"].dropna().unique().tolist()
meta = pd.DataFrame({"ticker": tks, "name": tks, "sector": ["Unknown"]*len(tks)})
con.executemany("INSERT OR IGNORE INTO meta(ticker,name,sector) VALUES(?,?,?)",
                meta.itertuples(index=False, name=None))
# Load prices.csv if present; otherwise synthesize small sample
if pathlib.Path("data/raw/prices.csv").exists():
    df = pd.read_csv("data/raw/prices.csv", parse_dates=["date"]).copy()
else:
    dates = pd.bdate_range("2022-01-03", periods=90)
    rng = np.random.default_rng(7)
    frames=[]
    for t in tks[:5]:
        r = rng.normal(0, 0.01, len(dates))
        price = 100*np.exp(np.cumsum(r))
        vol = rng.integers(1e5, 5e6, len(dates))
        frames.append(pd.DataFrame({"ticker": t, "date": dates,
                                    "adj_close": price, "volume": vol}))
    df = pd.concat(frames, ignore_index=True)
    df["log_return"] = np.log(df["adj_close"]).diff().fillna(0)
df["date"] = pd.to_datetime(df["date"]).dt.strftime("%Y-%m-%d")
df = df[["ticker","date","adj_close","volume","log_return"]].drop_duplicates(["ticker","d
con.executemany("INSERT OR REPLACE INTO prices(ticker,date,adj_close,volume,log_return) V
                df.itertuples(index=False, name=None))
con.commit()
```

```
    con.close()

# Connect and register SQRT (SQLite lacks STDDEV; we'll compute var and take sqrt)
con = sqlite3.connect(db_path)
con.create_function("SQRT", 1, lambda x: math.sqrt(x) if x is not None and x>=0 else None)
print("SQLite version:", sqlite3.sqlite_version)
```

### 7.4.2 1) LAG & LEAD (no leakage in features)

```
import pandas as pd

sql = """
SELECT ticker, date,
       log_return AS r,
       LAG(log_return,1) OVER (PARTITION BY ticker ORDER BY date) AS lag1,
       LAG(log_return,2) OVER (PARTITION BY ticker ORDER BY date) AS lag2,
       LEAD(log_return,1) OVER (PARTITION BY ticker ORDER BY date) AS r_tplus1  -- label canc
FROM prices
WHERE date BETWEEN ? AND ?
ORDER BY ticker, date
LIMIT 20;
"""
df = pd.read_sql_query(sql, con, params=["2022-03-01","2022-06-30"])
df.head(10)
```

**Teaching notes:**

- `lag1/lag2` are **safe features at time $t$** (depend on $t$-1).
- `r_tplus1` is a **label**; never include in features.

### 7.4.3 2) Named `WINDOW` + rolling mean/variance (20-row window)

```
sql = """
SELECT
  ticker, date, log_return AS r,
  AVG(log_return) OVER w AS roll_mean_20,
  AVG(log_return*log_return) OVER w
    - (AVG(log_return) OVER w)*(AVG(log_return) OVER w) AS roll_var_20
FROM prices
```

```
WINDOW w AS (
  PARTITION BY ticker
  ORDER BY date
  ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
)
WHERE date BETWEEN ? AND ?
ORDER BY ticker, date
LIMIT 20;
"""
roll = pd.read_sql_query(sql, con, params=["2022-03-01","2022-06-30"])
roll.head(10)
```

Compute rolling **std** and **z-score** in pandas (since we registered `SQRT`, we could also do it in SQL; here we'll do it in pandas for clarity):

```
roll["roll_std_20"] = (roll["roll_var_20"].clip(lower=0)).pow(0.5)
roll["zscore_20"] = (roll["r"] - roll["roll_mean_20"]) / roll["roll_std_20"].replace(0, pd.N
roll.head(5)
```

### 7.4.4 3) Top-k absolute moves per ticker with `ROW_NUMBER`

```
sql = """
WITH ranked AS (
  SELECT
    ticker, date, log_return,
    ABS(log_return) AS abs_move,
    ROW_NUMBER() OVER (
      PARTITION BY ticker
      ORDER BY ABS(log_return) DESC
    ) AS rn
  FROM prices
  WHERE date BETWEEN ? AND ?
)
SELECT * FROM ranked WHERE rn <= 3
ORDER BY ticker, rn;
"""
topk = pd.read_sql_query(sql, con, params=["2022-01-01","2025-08-01"])
topk.head(15)
```

### 7.4.5 4) Build a features DataFrame directly from SQL and save to Parquet

We'll assemble lags and rolling stats in one query using a named window `w20`:

```
sql = """
SELECT
  ticker, date,
  log_return AS r_1d,
  LAG(log_return,1) OVER (PARTITION BY ticker ORDER BY date) AS lag1,
  LAG(log_return,2) OVER (PARTITION BY ticker ORDER BY date) AS lag2,
  LAG(log_return,3) OVER (PARTITION BY ticker ORDER BY date) AS lag3,
  AVG(log_return) OVER w20 AS roll_mean_20,
  AVG(log_return*log_return) OVER w20
    - (AVG(log_return) OVER w20)*(AVG(log_return) OVER w20) AS roll_var_20
FROM prices
WINDOW w20 AS (
  PARTITION BY ticker
  ORDER BY date
  ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
)
WHERE date BETWEEN ? AND ?
ORDER BY ticker, date;
"""
features_sql = pd.read_sql_query(sql, con, params=["2019-01-01","2025-08-01"])
features_sql["roll_std_20"] = (features_sql["roll_var_20"].clip(lower=0)).pow(0.5)
features_sql["zscore_20"] = (features_sql["r_1d"] - features_sql["roll_mean_20"]) / features_

# Drop first 2-3 rows per ticker where lags are null
features_sql = (features_sql
                .sort_values(["ticker","date"])
                .groupby("ticker", group_keys=False)
                .apply(lambda g: g.iloc[3:]))

# Save
pathlib.Path("data/processed").mkdir(parents=True, exist_ok=True)
features_sql.to_parquet("data/processed/features_sql.parquet", index=False)
features_sql.head()
```

### 7.4.6 5) `EXPLAIN QUERY PLAN` sanity & index usage

```
plan = pd.read_sql_query("""
EXPLAIN QUERY PLAN
SELECT ticker, date,
       LAG(log_return,1) OVER (PARTITION BY ticker ORDER BY date) AS lag1
FROM prices
WHERE date BETWEEN ? AND ?
ORDER BY ticker, date;
""", con, params=["2022-01-01","2025-08-01"])
plan
```

> **Interpretation tip:** You should see use of the `idx_prices_date` or `PRIMARY KEY` (depending on the optimizer). If you often filter by `(ticker, date)`, consider a composite index: `CREATE INDEX IF NOT EXISTS idx_prices_ticker_date ON prices(ticker, date);` (We'll include that in homework.)

### 7.4.7 6) Save lab outputs

```
pathlib.Path("reports").mkdir(exist_ok=True)
features_sql.head(100).to_csv("reports/sql_window_demo.csv", index=False)
topk.to_csv("reports/top3_abs_moves_per_ticker.csv", index=False)
print("Wrote reports/sql_window_demo.csv and reports/top3_abs_moves_per_ticker.csv")
con.close()
```

---

## 7.5 Wrap-up (10 min)

- Window functions = **per-row context** (lags, rolling stats, top-k per group) with **no row collapse**.
- Prefer `ROWS BETWEEN N PRECEDING AND CURRENT ROW` to express true rolling windows.
- **No leakage**: only use `LAG` for features; `LEAD` is for labels.
- Use `pandas.read_sql_query` to **push computation into SQL** and bring back tidy frames.
- Indexes matter; check `EXPLAIN QUERY PLAN`, and add `(ticker, date)` index when filtering by both.

---

## 7.6 Homework (due before Session 9)

**Goal:** Productionize SQL-side feature engineering and performance basics. You will (A) create a reusable SQL file that defines features using windows, (B) write a small Python runner that writes **`data/processed/features_sql.parquet`**, (C) add a composite index, and (D) produce two small reports.

### 7.6.1 Part A — `sql/features_window.sql` (reusable)

Create `sql/features_window.sql`:

```
-- sql/features_window.sql
-- Rolling features and lags built with window functions.
WITH base AS (
  SELECT
    ticker, date, log_return AS r_1d
  FROM prices
  WHERE date BETWEEN ? AND ?  -- placeholders: start, end
)
SELECT
  ticker, date, r_1d,
  LAG(r_1d,1) OVER (PARTITION BY ticker ORDER BY date) AS lag1,
  LAG(r_1d,2) OVER (PARTITION BY ticker ORDER BY date) AS lag2,
  LAG(r_1d,3) OVER (PARTITION BY ticker ORDER BY date) AS lag3,
  AVG(r_1d) OVER w20 AS roll_mean_20,
  AVG(r_1d*r_1d) OVER w20 - (AVG(r_1d) OVER w20)*(AVG(r_1d) OVER w20) AS roll_var_20
FROM base
WINDOW w20 AS (
  PARTITION BY ticker
  ORDER BY date
  ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
)
ORDER BY ticker, date;
```

### 7.6.2 Part B — Runner: `scripts/build_features_sql.py`

```
# scripts/build_features_sql.py
#!/usr/bin/env python
import argparse, sqlite3, pandas as pd, numpy as np, math
```

```
from pathlib import Path

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--db", default="data/prices.db")
    ap.add_argument("--sqlfile", default="sql/features_window.sql")
    ap.add_argument("--start", default="2019-01-01")
    ap.add_argument("--end",   default="2025-08-01")
    ap.add_argument("--out",   default="data/processed/features_sql.parquet")
    ap.add_argument("--drop-head", type=int, default=3, help="Drop first N rows per ticker (d
    args = ap.parse_args()

    Path(args.out).parent.mkdir(parents=True, exist_ok=True)
    con = sqlite3.connect(args.db)
    con.create_function("SQRT", 1, lambda x: math.sqrt(x) if x is not None and x>=0 else None

    sql = Path(args.sqlfile).read_text()
    df = pd.read_sql_query(sql, con, params=[args.start, args.end])

    # Finish std & z-score in pandas
    df["roll_std_20"] = (df["roll_var_20"].clip(lower=0)).pow(0.5)
    df["zscore_20"] = (df["r_1d"] - df["roll_mean_20"]) / df["roll_std_20"].replace(0, pd.NA)

    # Drop first N rows per ticker where lags are NaN
    df = (df.sort_values(["ticker","date"])
            .groupby("ticker", group_keys=False)
            .apply(lambda g: g.iloc[args.drop_head:]))

    df.to_parquet(args.out, index=False)
    print("Wrote", args.out, "rows:", len(df))

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

Make it executable:

```
import os, stat, pathlib
p = pathlib.Path("scripts/build_features_sql.py")
os.chmod(p, os.stat(p).st_mode | stat.S_IEXEC)
print("Ready:", p)
```

### 7.6.3 Part C — Add a composite index and verify plan

1. Create `sql/add_index_ticker_date.sql`:

```
-- sql/add_index_ticker_date.sql
CREATE INDEX IF NOT EXISTS idx_prices_ticker_date ON prices(ticker, date);
```

2. Runner to apply it (or just execute once in a notebook):

```
import sqlite3, pathlib
con = sqlite3.connect("data/prices.db")
con.executescript(pathlib.Path("sql/add_index_ticker_date.sql").read_text())
con.close()
print("Index created: idx_prices_ticker_date")
```

3. Capture the query plan to a text file:

```
import sqlite3, pandas as pd, pathlib
con = sqlite3.connect("data/prices.db")
plan = pd.read_sql_query("""
EXPLAIN QUERY PLAN
SELECT ticker, date, LAG(log_return,1) OVER (PARTITION BY ticker ORDER BY date)
FROM prices
WHERE ticker = ? AND date BETWEEN ? AND ?
ORDER BY date;
""", con, params=["AAPL","2022-01-01","2025-08-01"])
pathlib.Path("reports").mkdir(exist_ok=True)
plan.to_csv("reports/query_plan_lag1.csv", index=False)
con.close()
print("Wrote reports/query_plan_lag1.csv")
```

### 7.6.4 Part D — Produce two small reports

1. **Top-k per ticker** (k=5) as CSV:

```
import sqlite3, pandas as pd, pathlib
con = sqlite3.connect("data/prices.db")
sql = """
WITH ranked AS (
  SELECT ticker, date, log_return, ABS(log_return) AS abs_move,
         ROW_NUMBER() OVER (PARTITION BY ticker ORDER BY ABS(log_return) DESC) AS rn
```

```
  FROM prices
  WHERE date BETWEEN ? AND ?
)
SELECT * FROM ranked WHERE rn <= 5 ORDER BY ticker, rn;
"""
df = pd.read_sql_query(sql, con, params=["2019-01-01","2025-08-01"])
pathlib.Path("reports").mkdir(exist_ok=True)
df.to_csv("reports/top5_abs_moves_per_ticker.csv", index=False)
con.close()
print("Wrote reports/top5_abs_moves_per_ticker.csv")
```

2. **Features via SQL** saved to Parquet:

```
!python scripts/build_features_sql.py --start 2019-01-01 --end 2025-08-01 --out data/processe
```

### 7.6.5 Part E — Makefile targets (optional but recommended)

Append to your `Makefile`:

```
DB := data/prices.db
FEATS_SQL := data/processed/features_sql.parquet

.PHONY: features-sql add-index plan
features-sql: $(FEATS_SQL) ## Build features using SQL windows
$(FEATS_SQL): scripts/build_features_sql.py sql/features_window.sql $(DB)
\tpython scripts/build_features_sql.py --db $(DB) --sqlfile sql/features_window.sql --start $

add-index: ## Create composite (ticker,date) index
\tsqlite3 $(DB) < sql/add_index_ticker_date.sql

plan: ## Save a sample EXPLAIN QUERY PLAN to reports/
\tpython - << 'PY'
import sqlite3, pandas as pd, os
con = sqlite3.connect("data/prices.db")
df = pd.read_sql_query(\"\"\"\nEXPLAIN QUERY PLAN\nSELECT ticker, date, LAG(log_return,1) OVE
os.makedirs("reports", exist_ok=True)
df.to_csv("reports/query_plan_lag1.csv", index=False)
con.close()
print("Wrote reports/query_plan_lag1.csv")
PY
```

Run:

```
%%bash
set -euo pipefail
cd "/content/drive/MyDrive/dspt25/unified-stocks-teamX"
make add-index
make features-sql
make plan
```

### 7.6.6 Submission checklist (pass/revise)

- `sql/features_window.sql` present; uses `WINDOW w20` and `LAG` for 1–3 lags.
- `scripts/build_features_sql.py` runs and writes `data/processed/features_sql.parquet`.
- Composite index created (`idx_prices_ticker_date`).
- `reports/top5_abs_moves_per_ticker.csv` and `reports/query_plan_lag1.csv` generated.
- (Optional) `Makefile` updated with `features-sql`, `add-index`, `plan`.

---

## 7.7 Instructor checklist (before class)

- Test the in-class lab once in a fresh Colab runtime; ensure SQLite version 3.25 (window functions).
- If a student's runtime is older (rare), advise upgrading Colab or running the Python fallback path.
- Keep one slide showing the difference between **`ROWS BETWEEN 19 PRECEDING AND CURRENT ROW`** and off-by-one mistakes.

## 7.8 Emphasize while teaching

- **No leakage**: features must come from `LAG`, not `LEAD`.
- Rolling stats with windows are **declarative and fast**; avoid reinventing in pandas unless needed.
- Use `WINDOW` names to reduce errors and duplication.
- Check query plans and add indexes purposefully.

**Next up (Session 9):** Finance-specific evaluation & leakage control — walk-forward splits, embargo, and regime-aware error analysis.

# 8 Session 9 — Cleaning, Joins, and Parquet

Below is a complete lecture package for **Session 9 — Cleaning, Joins, and Parquet** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. You'll clean & merge multi-ticker data, standardize dtypes (including **pandas nullable ints** and **categoricals**), and write tidy **Parquet**—including a partitioned-by-ticker dataset.

> **Assumptions:** Same Drive-mounted repo (e.g., `unified-stocks-teamX`) as prior sessions. Your raw prices live under `data/raw/` (either a single `prices.csv` or multiple CSVs). The lab includes a safe fallback (small synthetic dataset) if raw files are missing.

---

## 8.1 Session 9 — Cleaning, Joins, Parquet (75 min)

### 8.1.1 Learning goals

By the end of class, students can:

1. Use `merge`, `assign`, and `pipe` to write clean, testable data-wrangling code.
2. Choose **sensible dtypes** for analytics and storage: `category`, pandas **nullable integers** (`Int64`, `Int32`, …), and `string`.
3. Write **Parquet** with compression and **read it back**; understand **partitioning by ticker** and how to filter efficiently.

---

## 8.2 Agenda (75 min)

- **(10 min)** Slides: tidy schema; joins (`merge`), `assign`, `pipe` patterns
- **(10 min)** Slides: dtypes—`category`, `string`, **nullable ints** (`Int64`), `float32` vs `float64`
- **(10 min)** Slides: Parquet vs CSV; compression; partitioning; schema preservation
- **(35 min) In-class lab**: clean & join → set dtypes → write `data/processed/prices.parquet` and **partitioned** dataset by ticker
- **(10 min)** Wrap-up & homework briefing

---

## 8.3 Slides / talking points (paste these bullets into your deck)

### 8.3.1 Tidy schema for price data

- **One row = one ticker-day**.
- Minimal columns (snake_case): `date (datetime64[ns])`, `ticker (category)`, `open/high/low/close/adj_close (float32/64)`, `volume (Int64)`.
- Optional metadata (from a separate table): `name (string)`, `sector (category)`.

### 8.3.2 Idiomatic pandas: `merge`, `assign`, `pipe`

- **`merge`**: combine frames by keys (e.g., `prices` left join `tickers`).
- **`assign`**: add/transform columns without breaking the chain: `df = df.assign(adj_close=lambda d: d['adj_close'].fillna(d['close']))`.
- **`pipe`**: compose small, testable transforms: `df = (raw.pipe(standardize_columns).pipe(clean_price meta=meta))`.

### 8.3.3 Dtypes that help

- **Categorical** (`category`): compact & fast for low-cardinality strings (`ticker`, `sector`).
- **Nullable integers** (`Int64`, `Int32`): keep **missing values** and integer semantics (`volume`).
- **String** (`string[python]`): consistent string semantics (avoid `object`).
- **Floats**: `float32` can halve memory, but consider numeric precision.

### 8.3.4 Parquet: why & how

- **Columnar**, compressed, preserves schema better than CSV.
- **Filters & projection**: read only needed columns/rows (esp. with **partitioned datasets**).
- Partitioning by `ticker/` yields fast reads of a subset (e.g., a single ticker).
- Typical settings: engine=`pyarrow`, compression=`zstd` or `snappy`.

---

## 8.4 In-class lab (35 min)

Run each block as its **own Colab cell**. Replace `REPO_NAME` to match your repo.

### 8.4.1 0) Setup: mount Drive, cd into repo, ensure folders

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

#  Change this to your repo folder name
REPO_NAME  = "unified-stocks-teamX"
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, sys, glob, pandas as pd, numpy as np
pathlib.Path(BASE_DIR).mkdir(parents=True, exist_ok=True)
assert pathlib.Path(REPO_DIR).exists(), "Repo not found. Clone it first (Session 2/3)."
os.chdir(REPO_DIR)
for p in ["data/raw", "data/static", "data/processed"]:
    pathlib.Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())
```

### 8.4.2 1) Locate raw price files (CSV) or create a fallback

```
from pathlib import Path
import pandas as pd, numpy as np, datetime as dt
```

```python
raw_candidates = []
if Path("data/raw/prices.csv").exists():
    raw_candidates = ["data/raw/prices.csv"]
else:
    raw_candidates = sorted(glob.glob("data/raw/prices*.csv")) or sorted(glob.glob("data/raw/

def _make_synthetic_prices():
    # Small 2-year synthetic daily prices for AAPL/MSFT/GOOGL
    tickers = ["AAPL","MSFT","GOOGL"]
    dates = pd.bdate_range("2022-01-03", periods=520, freq="B")
    rows = []
    rng = np.random.default_rng(0)
    for t in tickers:
        price = 100 + rng.normal(0, 1).cumsum()
        price = np.maximum(price, 1.0)
        vol = rng.integers(5e6, 2e7, size=len(dates))
        df = pd.DataFrame({
            "date": dates,
            "ticker": t,
            "open": price * (1 + rng.normal(0, 0.002, size=len(dates))),
            "high": price * (1 + rng.normal(0.003, 0.003, size=len(dates))).clip(min=1),
            "low":  price * (1 - np.abs(rng.normal(0.003, 0.003, size=len(dates)))),
            "close": price,
            "adj_close": price * (1 + rng.normal(0, 0.0005, size=len(dates))),
            "volume": vol
        })
        rows.append(df)
    out = pd.concat(rows, ignore_index=True)
    Path("data/raw").mkdir(parents=True, exist_ok=True)
    out.to_csv("data/raw/prices.csv", index=False)
    return ["data/raw/prices.csv"]

if not raw_candidates:
    print("No raw prices found; creating a small synthetic dataset...")
    raw_candidates = _make_synthetic_prices()

raw_candidates
```

### 8.4.3 2) Optional metadata (tickers table), or create a minimal one

```
from pathlib import Path
meta_path = Path("data/static/tickers.csv")
if meta_path.exists():
    meta = pd.read_csv(meta_path)
else:
    # Build a minimal metadata table from raw tickers
    tmp = pd.read_csv(raw_candidates[0])
    tickers = sorted(pd.unique(tmp["ticker"]))
    meta = pd.DataFrame({"ticker": tickers,
                         "name": tickers,
                         "sector": ["Unknown"]*len(tickers)})
    Path("data/static").mkdir(parents=True, exist_ok=True)
    meta.to_csv(meta_path, index=False)
meta.head()
```

### 8.4.4 3) Helpers: `standardize_columns`, `clean_prices`, `join_meta` (showing `merge/assign/pipe`)

```
import re

def standardize_columns(df: pd.DataFrame) -> pd.DataFrame:
    """Lowercase snake_case; repair common price column name variants."""
    def snake(s):
        s = re.sub(r"[^\w\s]", "_", s)
        s = re.sub(r"\s+", "_", s.strip().lower())
        s = re.sub(r"_+", "_", s)
        return s
    out = df.copy()
    out.columns = [snake(c) for c in out.columns]
    # Normalize known variants
    ren = {
        "adjclose":"adj_close", "adj_close_":"adj_close",
        "close_adj":"adj_close", "adj_close_close":"adj_close"
    }
    out = out.rename(columns={k:v for k,v in ren.items() if k in out.columns})
    # If no adj_close but close exists, create it
    if "adj_close" not in out and "close" in out:
        out = out.assign(adj_close=out["close"])
```

```python
    return out

def clean_prices(df: pd.DataFrame) -> pd.DataFrame:
    """Coerce dtypes, drop dupes, basic sanity checks; add minor derived fields."""
    cols = ["date","ticker","open","high","low","close","adj_close","volume"]
    keep = [c for c in cols if c in df.columns]
    out = df.loc[:, keep].copy()

    # Parse date, coerce numerics
    out["date"] = pd.to_datetime(out["date"], errors="coerce")
    for c in ["open","high","low","close","adj_close"]:
        if c in out: out[c] = pd.to_numeric(out[c], errors="coerce")
    if "volume" in out: out["volume"] = pd.to_numeric(out["volume"], errors="coerce")

    # Drop bad rows
    out = out.dropna(subset=["date","ticker","adj_close"])
    # Deduplicate by (ticker, date)
    out = out.sort_values(["ticker","date"])
    out = out.drop_duplicates(subset=["ticker","date"], keep="last")

    # Enforce dtypes
    if "volume" in out:
        out["volume"] = out["volume"].round().astype("Int64")  # nullable int
        out.loc[out["volume"] < 0, "volume"] = pd.NA
    # Use category for low-cardinality strings
    out["ticker"] = out["ticker"].astype("category")
    # Use consistent float dtype
    for c in ["open","high","low","close","adj_close"]:
        if c in out: out[c] = out[c].astype("float32")  # ok for teaching; change to float64

    # Quick sanity checks
    assert out[["ticker","date"]].duplicated().sum() == 0, "Duplicates remain"
    assert pd.api.types.is_datetime64_any_dtype(out["date"]), "date not datetime"
    return out.reset_index(drop=True)

def join_meta(prices: pd.DataFrame, meta: pd.DataFrame) -> pd.DataFrame:
    """Left join metadata; keep minimal meta columns; set dtypes."""
    keep_meta = [c for c in ["ticker","name","sector"] if c in meta.columns]
    meta2 = meta.loc[:, keep_meta].copy()
    # Make strings consistent and compact
    if "name" in meta2:   meta2["name"]   = meta2["name"].astype("string")
    if "sector" in meta2: meta2["sector"] = meta2["sector"].astype("category")
```

```
    out = prices.merge(meta2, on="ticker", how="left", validate="many_to_one")
    return out
```

### 8.4.5 4) Read, clean, and join all raw files using a pipeline

```
dfs = []
for path in raw_candidates:
    raw = pd.read_csv(path)
    tidy = (raw
            .pipe(standardize_columns)  # <- consistent names
            .pipe(clean_prices))        # <- dtypes and sanity checks
    dfs.append(tidy)

prices = pd.concat(dfs, ignore_index=True)
prices = prices.pipe(join_meta, meta=meta)

print("Preview:")
display(prices.head(3))
print("\nInfo:")
print(prices.info(memory_usage="deep"))
```

### 8.4.6 5) Save clean data to Parquet (single file) and partitioned by ticker

```
# Single-file Parquet
single_path = "data/processed/prices.parquet"
prices.to_parquet(single_path, engine="pyarrow", compression="zstd", index=False)
print("Wrote:", single_path)

# Partitioned dataset by ticker (directory with /ticker=.../)
part_dir = "data/processed/prices_by_ticker"
# pandas to_parquet supports partition_cols with pyarrow engine
try:
    prices.to_parquet(part_dir, engine="pyarrow", compression="zstd",
                      index=False, partition_cols=["ticker"])
    print("Wrote partitioned dataset:", part_dir)
except TypeError:
    # Fallback via pyarrow dataset API
    import pyarrow as pa, pyarrow.parquet as pq
```

```
    pa_tbl = pa.Table.from_pandas(prices, preserve_index=False)
    pq.write_to_dataset(pa_tbl, root_path=part_dir, partition_cols=["ticker"], compression="
    print("Wrote (fallback) partitioned dataset:", part_dir)
```

### 8.4.7 6) Read back efficiently: projection + simple filters

```
# 6a) Read a few columns from single-file Parquet
cols = ["ticker","date","adj_close","volume"]
df_small = pd.read_parquet("data/processed/prices.parquet", columns=cols)
df_small.head()

# 6b) Read one ticker from the partitioned dataset using pyarrow.dataset
import pyarrow.dataset as ds
dataset = ds.dataset("data/processed/prices_by_ticker", format="parquet", partitioning="hive"
# Choose a ticker present in the data
one_ticker = str(prices["ticker"].cat.categories[0])
flt = (ds.field("ticker") == one_ticker)
tbl = dataset.to_table(filter=flt, columns=["date","adj_close","volume"])
df_one = tbl.to_pandas()
df_one.head()
```

### 8.4.8 7) Persist a simple schema record for reproducibility

```
import json, pathlib
schema = {c: str(t) for c,t in prices.dtypes.items()}
pathlib.Path("data/processed").mkdir(parents=True, exist_ok=True)
with open("data/processed/prices_schema.json","w") as f:
    json.dump(schema, f, indent=2)
print("Wrote data/processed/prices_schema.json")
```

---

## 8.5 Wrap-up (10 min) — points to emphasize

- A **tidy schema** makes life easier downstream.
- Prefer **`category`** for `ticker`, **nullable ints** for `volume`.

- Use **merge** (left join) to attach metadata; use **assign** for clear column creation; compose steps with **pipe**.
- **Parquet** is compact and fast; **partition by ticker** for selective reads.

---

## 8.6 Homework (due before Session 10)

**Deliverable:** `data/processed/returns.parquet` (and optionally partitioned by ticker) containing:

- `date`, `ticker`
- `log_return` — daily log return $\log(\frac{\text{adj\_close}_t}{\text{adj\_close}_{t-1}})$
- `r_1d` — **next-day** log return (lead of `log_return`)
- `weekday` (0=Mon..6=Sun), `month` (1..12) — choose compact dtypes

### 8.6.1 Step-by-step code (Colab-friendly)

Run in your repo root after finishing the in-class lab.

```
import pandas as pd, numpy as np, pathlib

# 1) Read the single-file Parquet you wrote in class
prices_path = "data/processed/prices.parquet"
assert pathlib.Path(prices_path).exists(), "Missing processed/prices.parquet - finish the lal

prices = pd.read_parquet(prices_path)
# If ticker was written as object, you can re-cast to category:
if prices["ticker"].dtype != "category":
    prices["ticker"] = prices["ticker"].astype("category")

# 2) Sort and compute returns per ticker (no leakage)
prices = prices.sort_values(["ticker","date"]).reset_index(drop=True)

# Daily log return: log(adj_close_t / adj_close_{t-1})
prices["log_return"] = (prices.groupby("ticker")["adj_close"]
                        .apply(lambda s: np.log(s / s.shift(1))).reset_index(level=0, drop=T:

# Next-day return label r_1d = lead(log_return, 1)
prices["r_1d"] = (prices.groupby("ticker")["log_return"]
```

```
                   .shift(-1))

# 3) Calendar features
prices["weekday"] = prices["date"].dt.weekday.astype("int8")  # 0..6
prices["month"]   = prices["date"].dt.month.astype("int8")    # 1..12

# 4) Select output columns & dtypes
out = prices[["date","ticker","log_return","r_1d","weekday","month"]].copy()
out["ticker"] = out["ticker"].astype("category")

# 5) Save to Parquet
out_path = "data/processed/returns.parquet"
out.to_parquet(out_path, engine="pyarrow", compression="zstd", index=False)
print("Wrote:", out_path)

# 6) (Optional) also write a partitioned dataset by ticker
part_dir = "data/processed/returns_by_ticker"
try:
    out.to_parquet(part_dir, engine="pyarrow", compression="zstd",
                   index=False, partition_cols=["ticker"])
    print("Wrote partitioned dataset:", part_dir)
except TypeError:
    import pyarrow as pa, pyarrow.parquet as pq
    pq.write_to_dataset(pa.Table.from_pandas(out, preserve_index=False),
                        root_path=part_dir, partition_cols=["ticker"], compression="zstd")
    print("Wrote (fallback) partitioned dataset:", part_dir)
```

### 8.6.2 Quick self-check (run after saving)

```
import pandas as pd
r = pd.read_parquet("data/processed/returns.parquet")
assert {"date","ticker","log_return","r_1d","weekday","month"}.issubset(r.columns)
assert r["ticker"].dtype.name in ("category","CategoricalDtype"), "ticker should be categori
print("rows:", len(r), "| tickers:", r["ticker"].nunique())
r.head()
```

### 8.6.3 (Optional) Extra credit

- Add **year** (Int16) and **is_month_end** (BooleanDtype): `r["year"] = r["date"].dt.year.astype("Int16` `r["is_month_end"] = r["date"].dt.is_month_end.astype("boolean")`

- Compare file sizes: CSV vs Parquet vs Parquet (`zstd` vs `snappy`).

### 8.6.4 Submission checklist (pass/revise)

- `data/processed/returns.parquet` exists and contains the required columns.
- `ticker` is **categorical**; `weekday`/`month` are compact ints.
- `r_1d` is a **lead** of `log_return` (next-day), not the same-day return.
- You can read it back without errors.

---

## 8.7 Instructor notes / gotchas to watch for

- **Nullable ints**: `astype("Int64")` keeps NAs; plain `int64` will fail if NAs exist.
- **Categoricals & partitions**: When reading partitioned Parquet, `ticker` may come back as `object`. Re-cast to `category` after read if needed.
- **Compression choice**: `zstd` gives good ratio/speed; `snappy` is more ubiquitous.
- **Precision**: `float32` is fine for teaching; for production finance, consider `float64` and explicit rounding.

---

### 8.7.1 Optional (for your Makefile later)

Add quick targets:

```
.PHONY: prices-parquet returns-parquet
prices-parquet:  ## Clean raw prices and save processed Parquet(s)
\tpython - <<'PY'
import pandas as pd, glob, pathlib, numpy as np, re, json
from pathlib import Path
# (Paste the functions from the lab: standardize_columns, clean_prices, join_meta)
# Then read raw -> clean -> write parquet as in the lab
PY

returns-parquet: ## Build returns.parquet with r_1d + calendar features
\tpython - <<'PY'
import pandas as pd, numpy as np
p="data/processed/prices.parquet"; r=pd.read_parquet(p).sort_values(["ticker","date"])
```

```
r["log_return"]=r.groupby("ticker")["adj_close"].apply(lambda s: np.log(s/s.shift(1))).reset_
r["r_1d"]=r.groupby("ticker")["log_return"].shift(-1)
r["weekday"]=r["date"].dt.weekday.astype("int8"); r["month"]=r["date"].dt.month.astype("int8"
r[["date","ticker","log_return","r_1d","weekday","month"]].to_parquet("data/processed/returns
print("Wrote data/processed/returns.parquet")
PY
```

You now have a clean, tidy Parquet foundation the later sessions (evaluation & modeling) can rely on.

# 9 Rolling Windows, Resampling, and Leakage-Safe Features

Fantastic—thanks for pasting the full course plan. Below are **three complete 75-minute lecture packages** for **Sessions 10, 11, and 12**, each with: a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. They assume you completed Session 9 and have `data/processed/prices.parquet` and `data/processed/returns.parquet`. Every lab includes fallbacks so you can run them even if files are missing.

> **Educational use only — not trading advice.** Python-only. Colab + Drive assumed. If you don't already have the repo and folders used below, the first cells create them.

---

# 10 Session 10 — Rolling Windows, Resampling, and Leakage-Safe Features (75 min)

#### 10.0.1 Learning goals

By the end of class students can:

1. Use `groupby('ticker')` with `shift`, `rolling`, `expanding`, and `ewm` to engineer features **without leakage**.
2. Resample safely (daily → weekly/monthly) and understand how to aggregate OHLC + volume.
3. Produce a tidy `features_v1.parquet` with sensible dtypes.

---

## 10.1 Agenda (75 min)

- **(10 min)** Slides: leakage-free features; lags vs rolling; resampling patterns
- **(10 min)** Slides: common pitfalls (min_periods, alignment, mixed frequencies)
- **(35 min) In-class lab**: load returns → build features → (optional) weekly aggregates → write `features_v1.parquet`
- **(10 min)** Wrap-up + homework brief
- **(10 min)** Buffer

---

## 10.2 Slide talking points

**Feature timing = everything**

- Predict $r_{t+1}$ using info up to and including **time t**.

- **Rule:** compute any rolling stat at $t$ from data $\leq t$, then **shift by 1** if that stat includes the current target variable.

**Core pandas patterns**

- **Lags:** `s.shift(k)` (past), never negative shifts.
- **Rolling:** `s.rolling(W, min_periods=W).agg(...)` and then **no extra shift** if the rolling window ends at $t$.
- **Expanding:** long-memory features (e.g., expanding mean).
- **EWM:** `s.ewm(span=W, adjust=False).mean()` for decayed memory.

**Resampling safely**

- Use `groupby('ticker').resample('W-FRI', on='date')` then aggregate:
  - OHLC: `first/open`, `max/high`, `min/low`, `last/adj_close`
  - Volume: `sum`
  - Returns: compound via `np.log(prod(1+r))` or sum of log returns.

**Dtypes**

- `ticker` = `category`; calendar ints `int8`; features `float32` (fine for class).

---

## 10.3 In-class lab (Colab-friendly)

Run each block as its own cell. Adjust `REPO_NAME` as needed.

### 10.3.1 0) Setup

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"   # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, numpy as np, pandas as pd
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
```

```
for p in ["data/raw","data/processed","reports","scripts","tests"]:
    pathlib.Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())
```

### 10.3.2 1) Load inputs or build small fallbacks

```
from pathlib import Path
rng = np.random.default_rng(0)

# Fallback synthetic if missing
def make_synth_prices():
    dates = pd.bdate_range("2022-01-03", periods=300)
    frames=[]
    for tkr in ["AAPL","MSFT","GOOGL","AMZN","NVDA"]:
        base = 100 + rng.normal(0,1, size=len(dates)).cumsum()
        d = pd.DataFrame({
            "date": dates, "ticker": tkr,
            "adj_close": np.maximum(base, 1.0).astype("float32"),
            "volume": rng.integers(1e6, 5e6, size=len(dates)).astype("int64")
        })
        frames.append(d)
    prices = pd.concat(frames, ignore_index=True)
    prices["ticker"] = prices["ticker"].astype("category")
    prices.to_parquet("data/processed/prices.parquet", index=False)
    return prices

ppath = Path("data/processed/prices.parquet")
rpath = Path("data/processed/returns.parquet")

if ppath.exists():
    prices = pd.read_parquet(ppath)
else:
    prices = make_synth_prices()

# Build returns if missing (from Session 9 logic)
if rpath.exists():
    returns = pd.read_parquet(rpath)
else:
    df = prices.sort_values(["ticker","date"]).copy()
    df["log_return"] = (df.groupby("ticker")["adj_close"]
```

```
                        .apply(lambda s: np.log(s/s.shift(1))).reset_index(level=0, drop=True
    df["r_1d"] = df.groupby("ticker")["log_return"].shift(-1)
    df["weekday"] = df["date"].dt.weekday.astype("int8")
    df["month"]   = df["date"].dt.month.astype("int8")
    returns = df[["date","ticker","log_return","r_1d","weekday","month"]].copy()
    returns["ticker"] = returns["ticker"].astype("category")
    returns.to_parquet("data/processed/returns.parquet", index=False)

prices.head(3), returns.head(3)
```

### 10.3.3 2) Rolling, lag, expanding, ewm features (no leakage)

```
def build_features(ret: pd.DataFrame, windows=(5,10,20), add_rsi=True):
    g = ret.sort_values(["ticker","date"]).groupby("ticker", group_keys=False)
    out = ret.copy()

    # Lags of log_return (past info)
    for k in [1,2,3]:
        out[f"lag{k}"] = g["log_return"].shift(k)

    # Rolling mean/std and z-score of returns using past W days **including today**,
    # which is fine because target is r_{t+1}. No extra shift needed.
    for W in windows:
        rm = g["log_return"].rolling(W, min_periods=W).mean()
        rsd= g["log_return"].rolling(W, min_periods=W).std()
        out[f"roll_mean_{W}"] = rm.reset_index(level=0, drop=True)
        out[f"roll_std_{W}"]  = rsd.reset_index(level=0, drop=True)
        out[f"zscore_{W}"]    = (out["log_return"] - out[f"roll_mean_{W}"]) / (out[f"roll_st

    # Expanding stats (from start to t): long-memory
    out["exp_mean"] = g["log_return"].expanding(min_periods=20).mean().reset_index(level=0, d
    out["exp_std"]  = g["log_return"].expanding(min_periods=20).std().reset_index(level=0, di

    # Exponential weighted (decayed memory)
    for W in [10,20]:
        out[f"ewm_mean_{W}"] = g["log_return"].apply(lambda s: s.ewm(span=W, adjust=False).me
        out[f"ewm_std_{W}"]  = g["log_return"].apply(lambda s: s.ewm(span=W, adjust=False).s

    # Optional RSI(14) using returns sign proxy (toy version)
    if add_rsi:
```

```
        def rsi14(s):
            delta = s.diff()
            up = delta.clip(lower=0).ewm(alpha=1/14, adjust=False).mean()
            dn = (-delta.clip(upper=0)).ewm(alpha=1/14, adjust=False).mean()
            rs = up / (dn + 1e-12)
            return 100 - (100 / (1 + rs))
        out["rsi_14"] = g["adj_close"].apply(rsi14) if "adj_close" in out else g["log_return"

    # Cast dtypes
    for c in out.columns:
        if c not in ["date","ticker","weekday","month"] and pd.api.types.is_float_dtype(out[c
            out[c] = out[c].astype("float32")
    out["ticker"] = out["ticker"].astype("category")
    return out

# Merge adj_close and volume into returns (if not already)
ret2 = returns.merge(prices[["ticker","date","adj_close","volume"]], on=["ticker","date"], ho
features = build_features(ret2, windows=(5,10,20), add_rsi=True)
features.head(5)
```

### 10.3.4 3) (Optional) Weekly resampling demo (OHLCV + returns)

```
# Safe weekly resample per ticker, aggregating OHLCV and log returns
def weekly_ohlcv(df):
    df = df.sort_values(["ticker","date"]).copy()
    df["date"] = pd.to_datetime(df["date"])
    res=[]
    for tkr, g in df.groupby("ticker"):
        wk = (g.resample("W-FRI", on="date")
              .agg({"adj_close":"last","volume":"sum"}).dropna().reset_index())
        wk["ticker"] = tkr
        # Weekly log return = log(adj_close_t / adj_close_{t-1})
        wk = wk.sort_values("date")
        wk["wk_log_return"] = np.log(wk["adj_close"]/wk["adj_close"].shift(1))
        res.append(wk)
    return pd.concat(res, ignore_index=True)

weekly = weekly_ohlcv(prices[["ticker","date","adj_close","volume"]])
weekly.head(5)
```

### 10.3.5 4) Save `features_v1.parquet` (+ optional partition by ticker)

```
# Select a compact set to start with
keep = ["date","ticker","log_return","r_1d","weekday","month",
        "lag1","lag2","lag3",
        "roll_mean_5","roll_std_5","zscore_5",
        "roll_mean_10","roll_std_10","zscore_10",
        "roll_mean_20","roll_std_20","zscore_20",
        "ewm_mean_10","ewm_std_10","ewm_mean_20","ewm_std_20",
        "exp_mean","exp_std","rsi_14","adj_close","volume"]

keep = [c for c in keep if c in features.columns]
fv1 = features.loc[:, keep].dropna().sort_values(["ticker","date"]).reset_index(drop=True)
fv1["weekday"] = fv1["weekday"].astype("int8")
fv1["month"]   = fv1["month"].astype("int8")
fv1["ticker"]  = fv1["ticker"].astype("category")

fv1_path = "data/processed/features_v1.parquet"
fv1.to_parquet(fv1_path, compression="zstd", index=False)
print("Wrote:", fv1_path, "| rows:", len(fv1), "| cols:", len(fv1.columns))

# Optional partition
part_dir = "data/processed/features_v1_by_ticker"
try:
    fv1.to_parquet(part_dir, compression="zstd", index=False, engine="pyarrow", partition_co
    print("Wrote partitioned:", part_dir)
except TypeError:
    print("Partition writing skipped (engine missing).")
```

---

## 10.4 Wrap-up (what to emphasize)

- For **next-day** targets $r_{t+1}$, rolling stats up to **t** are fine; never use future rows.
- Be explicit about **`min_periods`** to avoid unstable early rows.
- Keep features small and typed; document your cookbook in the repo.

---

## 10.5 Homework (due before Session 11)

**Goal:** Add an **automated leakage check** and re-run feature build.

### 10.5.1 A. Script: `scripts/build_features_v1.py`

```python
#!/usr/bin/env python
import numpy as np, pandas as pd, pathlib
def build():
    p = pathlib.Path("data/processed/returns.parquet")
    if not p.exists(): raise SystemExit("Missing returns.parquet - finish Session 9.")
    prices = pd.read_parquet("data/processed/prices.parquet")
    ret = pd.read_parquet(p)
    ret2 = ret.merge(prices[["ticker","date","adj_close","volume"]], on=["ticker","date"], ho
    # (Paste the build_features() from class)
    # ...
    fv1 = build_features(ret2)
    keep = ["date","ticker","log_return","r_1d","weekday","month",
            "lag1","lag2","lag3","roll_mean_20","roll_std_20","zscore_20",
            "ewm_mean_20","ewm_std_20","exp_mean","exp_std","adj_close","volume"]
    keep = [c for c in keep if c in fv1.columns]
    fv1 = fv1[keep].dropna().sort_values(["ticker","date"])
    fv1.to_parquet("data/processed/features_v1.parquet", compression="zstd", index=False)
    print("Wrote data/processed/features_v1.parquet", fv1.shape)
if __name__ == "__main__":
    build()
```

Make executable:

```bash
%%bash
chmod +x scripts/build_features_v1.py
python scripts/build_features_v1.py
```

### 10.5.2 B. Test: `tests/test_no_lookahead.py`

```python
import pandas as pd, numpy as np

def test_features_no_lookahead():
```

```
    df = pd.read_parquet("data/processed/features_v1.parquet").sort_values(["ticker","date"])
    # For each ticker, recompute roll_mean_20 with an independent method and compare
    for tkr, g in df.groupby("ticker"):
        s = g["log_return"]
        rm = s.rolling(20, min_periods=20).mean()
        # Our feature should equal this rolling mean (within tol)
        if "roll_mean_20" in g:
            assert np.allclose(g["roll_mean_20"].values, rm.values, equal_nan=True, atol=1e-
        # r_1d must be the **lead** of log_return
        assert g["r_1d"].shift(1).iloc[21:].equals(g["log_return"].iloc[21:])
```

Run:

```
%%bash
pytest -q
```

---

# 11 Session 11 — APIs with `requests`: Secrets, Retries, and Caching

# 12 Session 11 — APIs with `requests`: Secrets, Retries, and Caching (75 min)

#### 12.0.1 Learning goals

Students will be able to:

1. Call a REST API with `requests` + robust retry/backoff.
2. Manage secrets with `.env` and **never** commit keys.
3. Cache responses (file or SQLite) and align external series by date.
4. Save enriched data to SQLite and Parquet.

---

## 12.1 Agenda (75 min)

- **(10 min)** Slides: anatomy of a GET; query params; JSON; status codes
- **(10 min)** Slides: secrets (`python-dotenv`), file layout (`.env`, `.env.template`), `.gitignore`
- **(10 min)** Slides: retries and caching patterns; idempotent design
- **(35 min) In-class lab**: fetch **FRED VIX (VIXCLS)** + optional **FEDFUNDS** → cache → store in SQLite → join to daily features
- **(10 min)** Wrap-up + homework

---

## 12.2 Slide talking points

**Requests pattern**

- `Session` + `HTTPAdapter` + `Retry` → robust.
- Validate: status code, content type; guard against partial data.

**Secrets**

- `.env.template` committed; `.env` untracked.
- Load with `dotenv.load_dotenv()`. Access via `os.getenv("FRED_API_KEY")`.

**Caching**

- **File cache**: key by URL+params hash.
- **DB cache**: `cache (key TEXT PRIMARY KEY, value BLOB, fetched_at)`.

**Alignment**

- After download, **normalize to `date`** and join on date.
- Store to SQLite table with a composite key `(series_id, date)`.

---

## 12.3 In-class lab

### 12.3.1 0) Setup, folders, and templates

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, json, hashlib, time, sqlite3, pandas as pd, numpy as np
from pathlib import Path
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in [".cache/api","data","data/processed","data/raw"]:
    Path(p).mkdir(parents=True, exist_ok=True)

# .env template for secrets
Path(".env.template").write_text("FRED_API_KEY=\n")
# Ensure .gitignore has secrets & cache
gi = Path(".gitignore")
if gi.exists():
    gi_txt = gi.read_text()
else:
    gi_txt = ""
```

```
for line in [".env", ".cache/", "__pycache__/"]:
    if line not in gi_txt:
        gi_txt += ("\n" if not gi_txt.endswith("\n") else "") + line
gi.write_text(gi_txt)
print("Ready. Fill your FRED key in a local .env (do not commit).")
```

### 12.3.2 1) Robust GET with retry + file cache

```
import os, requests
from urllib3.util.retry import Retry
from requests.adapters import HTTPAdapter
from dotenv import load_dotenv

load_dotenv()  # reads .env if present

def session_with_retry(total=3, backoff=0.5):
    s = requests.Session()
    retry = Retry(total=total, backoff_factor=backoff, status_forcelist=[429,500,502,503,504]
    s.mount("https://", HTTPAdapter(max_retries=retry))
    s.headers.update({"User-Agent": "dspt-class/1.0 (+edu)"})
    return s

def cache_key(url, params):
    raw = url + "?" + "&".join(f"{k}={params[k]}" for k in sorted(params))
    return hashlib.sha1(raw.encode()).hexdigest()

def cached_get(url, params, ttl_hours=24):
    key = cache_key(url, params)
    path = Path(f".cache/api/{key}.json")
    if path.exists() and (time.time() - path.stat().st_mtime < ttl_hours*3600):
        return json.loads(path.read_text())
    s = session_with_retry()
    r = s.get(url, params=params, timeout=20)
    r.raise_for_status()
    data = r.json()
    path.write_text(json.dumps(data))
    return data
```

### 12.3.3 2) Fetch VIX (VIXCLS) and FEDFUNDS from FRED; store to SQLite

```
API_KEY = os.getenv("FRED_API_KEY", "").strip()
if not API_KEY:
    print("WARNING: No FRED_API_KEY in .env; continuing with unauthenticated request may fai

FRED_SERIES_URL = "https://api.stlouisfed.org/fred/series/observations"

def fred_series(series_id, start="2010-01-01", end=None):
    p = {"series_id":series_id, "api_key":API_KEY, "file_type":"json",
         "observation_start":start}
    if end is not None: p["observation_end"]=end
    data = cached_get(FRED_SERIES_URL, p, ttl_hours=24)
    obs = data.get("observations", [])
    df = pd.DataFrame(obs)[["date","value"]]
    df["date"] = pd.to_datetime(df["date"])
    df["value"] = pd.to_numeric(df["value"], errors="coerce")
    df["series_id"] = series_id
    return df.dropna()

vix = fred_series("VIXCLS", start="2015-01-01")       # CBOE VIX
fed = fred_series("FEDFUNDS", start="2015-01-01")     # Effective Fed Funds

# Write to SQLite
db = sqlite3.connect("data/prices.db")
db.execute("""CREATE TABLE IF NOT EXISTS macro_series(
    series_id TEXT NOT NULL, date TEXT NOT NULL, value REAL NOT NULL,
    PRIMARY KEY(series_id, date))""")
for df in [vix, fed]:
    df.to_sql("macro_series", db, if_exists="append", index=False)
db.commit(); db.close()

vix.head(), fed.head()
```

### 12.3.4 3) Join macro series to daily returns/features by `date`

```
# Load features (build if missing)
from pathlib import Path
fvpath = Path("data/processed/features_v1.parquet")
if not fvpath.exists():
```

```
    raise SystemExit("Missing features_v1.parquet - run Session 10 lab or homework.")

fv1 = pd.read_parquet(fvpath).sort_values(["ticker","date"])
macro = pd.concat([vix.rename(columns={"value":"vix"}).drop(columns="series_id"),
                   fed.rename(columns={"value":"fedfunds"}).drop(columns="series_id")], axis=
# Pivot macro wide
macro_wide = (pd.concat([
    vix.assign(var="vix").rename(columns={"value":"val"}),
    fed.assign(var="fedfunds").rename(columns={"value":"val"})
]) .pivot_table(index="date", columns="var", values="val").reset_index())

enriched = fv1.merge(macro_wide, on="date", how="left")
enriched[["vix","fedfunds"]] = enriched[["vix","fedfunds"]].astype("float32")
enriched.to_parquet("data/processed/features_v1_ext.parquet", compression="zstd", index=False
print("Wrote data/processed/features_v1_ext.parquet", enriched.shape)
enriched.head(5)
```

---

## 12.4 Wrap-up

- You built a **retrying, cached** API client, stored macro data in **SQLite**, and aligned it by date.
- Secrets live in `.env` (never committed).
- Enriched features are saved for modeling later.

---

## 12.5 Homework (due before Session 12)

**Goal:** Add **one more external series** (your choice) via FRED and keep everything cached and reproducible.

### 12.5.1 A. Script: `scripts/get_macro.py`

```
#!/usr/bin/env python
import os, json, time, hashlib, pandas as pd, sqlite3
from pathlib import Path
import requests
from urllib3.util.retry import Retry
from requests.adapters import HTTPAdapter
from dotenv import load_dotenv

load_dotenv()
API_KEY = os.getenv("FRED_API_KEY","").strip()
BASE = "https://api.stlouisfed.org/fred/series/observations"

def sess():
    s = requests.Session()
    s.headers.update({"User-Agent":"dspt-class/1.0"})
    s.mount("https://", HTTPAdapter(max_retries=Retry(total=3, backoff_factor=0.5,
                                                     status_forcelist=[429,500,502,503,504]
    return s

def ckey(url, params):
    raw = url + "?" + "&".join(f"{k}={params[k]}" for k in sorted(params))
    return hashlib.sha1(raw.encode()).hexdigest()

def cached_get(url, params, ttl=86400):
    key = ckey(url, params); p = Path(f".cache/api/{key}.json")
    if p.exists() and (time.time() - p.stat().st_mtime < ttl):
        return json.loads(p.read_text())
    r = sess().get(url, params=params, timeout=20); r.raise_for_status()
    data = r.json(); p.write_text(json.dumps(data)); return data

def fetch_series(series_id, start="2015-01-01"):
    if not API_KEY: raise SystemExit("Set FRED_API_KEY in .env")
    params = {"series_id":series_id, "api_key":API_KEY, "file_type":"json", "observation_sta
    data = cached_get(BASE, params)
    df = pd.DataFrame(data["observations"])[["date","value"]]
    df["date"] = pd.to_datetime(df["date"])
    df["value"] = pd.to_numeric(df["value"], errors="coerce")
    df["series_id"] = series_id
    return df.dropna()

def main(series_id):
    df = fetch_series(series_id)
```

```python
    con = sqlite3.connect("data/prices.db")
    con.execute("""CREATE TABLE IF NOT EXISTS macro_series(
        series_id TEXT, date TEXT, value REAL, PRIMARY KEY(series_id,date))""")
    df.to_sql("macro_series", con, if_exists="append", index=False)
    con.commit(); con.close()
    print(f"Stored {series_id}: {len(df)} rows")

if __name__ == "__main__":
    import argparse
    ap = argparse.ArgumentParser()
    ap.add_argument("--series-id", required=True)
    ap.add_argument("--start", default="2015-01-01")
    args = ap.parse_args()
    main(args.series_id)
```

Run example:

```bash
%%bash
chmod +x scripts/get_macro.py
python scripts/get_macro.py --series-id DGS10   # 10-Year Treasury Constant Maturity Rate
```

### 12.5.2 B. Enrich features with your new series

```python
import pandas as pd, sqlite3
fv = pd.read_parquet("data/processed/features_v1.parquet")
con = sqlite3.connect("data/prices.db")
macro = pd.read_sql_query("SELECT series_id, date, value FROM macro_series", con, parse_dates
con.close()
wide = macro.pivot_table(index="date", columns="series_id", values="value").reset_index()
out = fv.merge(wide, on="date", how="left")
out.to_parquet("data/processed/features_v1_ext.parquet", compression="zstd", index=False)
print("Wrote features_v1_ext.parquet with extra series:", out.shape)
```

### 12.5.3 C. Short test: `tests/test_macro_join.py`

```python
import pandas as pd
def test_enriched_has_macro():
    df = pd.read_parquet("data/processed/features_v1_ext.parquet")
```

```
    assert "date" in df and "ticker" in df
    assert df.filter(regex="^(VIXCLS|DGS10|FEDFUNDS)$").shape[1] >= 1
```

Run:

```
%%bash
pytest -q
```

---

# 13 Session 12 — HTML Scraping: Ethics & Resilience

# 14 Session 12 — HTML Scraping: Ethics & Resilience (75 min)

#### 14.0.1 Learning goals

Students will be able to:

1. Respect robots.txt and basic site etiquette (throttling, user-agent, caching).
2. Extract structured tables with **BeautifulSoup** and fall back to `pandas.read_html`.
3. Normalize scraped data (clean headers, dtypes, categories).
4. Save provenance and update a **data dictionary** for the repo.

---

## 14.1 Agenda (75 min)

- **(10 min)** Slides: ethics, robots, terms; caching, rate limits
- **(10 min)** Slides: stable selectors (ids, table headers), text cleanup, date parsing
- **(35 min) In-class lab**: scrape a static sector table (Wikipedia S&P 500 components), map to your tickers, save `data/static/sector_map.csv`; merge into `prices.parquet` if missing
- **(10 min)** Wrap-up + homework brief
- **(10 min)** Buffer

---

## 14.2 Slide talking points

**Ethics + resilience**

- **Check robots.txt**; identify disallow rules.
- Set a clear **User-Agent** and **sleep** between requests.
- Cache HTML locally; **don't hammer** sites.

- Expect structure to change; write **defensive** code.

**Parsing patterns**

- Prefer table selectors; use `read_html` for well-formed tables.
- Clean headers → snake_case; drop footnotes; trim whitespace.
- Normalize keys (e.g., ticker symbols: map `.` `-` if needed).

**Provenance**

- Save `source_url`, `fetched_at`, and a checksum alongside the CSV.

---

## 14.3 In-class lab

> We'll scrape **Wikipedia: List of S&P 500 companies** (static table). If blocked, we fall back to `pandas.read_html` or a small local stub.

### 14.3.1 0) Setup + robots check + HTML caching

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, requests, time, hashlib, pandas as pd, numpy as np
from bs4 import BeautifulSoup
from urllib.parse import urljoin
from datetime import datetime

os.chdir(REPO_DIR)
for p in [".cache/html","data/static","reports"]:
    pathlib.Path(p).mkdir(parents=True, exist_ok=True)

UA = {"User-Agent": "dspt-class/1.0 (+edu)"}
WIKI_URL = "https://en.wikipedia.org/wiki/List_of_S%26P_500_companies"
```

```
def allowed_by_robots(base, path="/wiki/"):
    r = requests.get(urljoin(base, "/robots.txt"), headers=UA, timeout=20)
    if r.status_code != 200: return True
    lines = r.text.splitlines()
    disallows = [ln.split(":")[1].strip() for ln in lines if ln.lower().startswith("disallow
    return all(not path.startswith(d) for d in disallows)

print("Robots allows /wiki/?", allowed_by_robots("https://en.wikipedia.org"))
```

### 14.3.2 1) Download (with cache) and parse the first big table

```
def get_html_cached(url, ttl_hours=24):
    key = hashlib.sha1(url.encode()).hexdigest()
    path = pathlib.Path(f".cache/html/{key}.html")
    if path.exists() and (time.time() - path.stat().st_mtime < ttl_hours * 3600):
        return path.read_text()
    r = requests.get(url, headers=UA, timeout=30)
    r.raise_for_status()
    path.write_text(r.text)
    time.sleep(1.0)  # be polite
    return r.text

html = get_html_cached(WIKI_URL)
soup = BeautifulSoup(html, "html.parser")

# Try soup table first; fallback to pandas.read_html
table = soup.find("table", {"id":"constituents"}) or soup.find("table", {"class":"wikitable"]
if table is not None:
    rows = []
    headers = [th.get_text(strip=True) for th in table.find("tr").find_all("th")]
    for tr in table.find_all("tr")[1:]:
        tds = [td.get_text(strip=True) for td in tr.find_all(["td","th"])]
        if len(tds) == len(headers):
            rows.append(dict(zip(headers, tds)))
    sp = pd.DataFrame(rows)
else:
    sp = pd.read_html(html)[0]

sp.head(3), sp.columns.tolist()
```

### 14.3.3 2) Clean + normalize + keep only ticker sector

```
import re
def snake(s):
    s = re.sub(r"[^\w\s]", "_", s)
    s = re.sub(r"\s+", "_", s.strip().lower())
    return re.sub(r"_+", "_", s)

sp.columns = [snake(c) for c in sp.columns]
cand_cols = [c for c in sp.columns if "symbol" in c or "security" in c or "sector" in c]
sp = sp.rename(columns={c:"symbol" for c in sp.columns if "symbol" in c or c=="ticker"})
sp = sp.rename(columns={c:"sector" for c in sp.columns if "sector" in c})
keep = [c for c in ["symbol","sector"] if c in sp.columns]
sp = sp[keep].dropna().drop_duplicates()
sp = sp.rename(columns={"symbol":"ticker"})
sp["ticker"] = sp["ticker"].str.strip()
sp["sector"] = sp["sector"].astype("category")

# Save with provenance
src = {"source_url": WIKI_URL, "fetched_at_utc": datetime.utcnow().isoformat()+"Z"}
sp.to_csv("data/static/sector_map.csv", index=False)
with open("data/static/sector_map.provenance.json","w") as f:
    import json; json.dump(src, f, indent=2)
print("Wrote data/static/sector_map.csv", sp.shape)
sp.head(5)
```

### 14.3.4 3) Merge sector mapping into prices if missing sector

```
from pathlib import Path
pp = Path("data/processed/prices.parquet")
if not pp.exists():
    raise SystemExit("Need prices.parquet (Session 9).")

prices = pd.read_parquet(pp)
if "sector" not in prices.columns or prices["sector"].isna().all():
    prices2 = prices.merge(sp, on="ticker", how="left")
    prices2["sector"] = prices2["sector"].astype("category")
    prices2.to_parquet("data/processed/prices.parquet", compression="zstd", index=False)
    print("Updated prices.parquet with sector column.")
```

```
else:
    print("Sector already present; no merge needed.")
```

---

## 14.4 Wrap-up

- You scraped a static table **politely** (robots, throttle, cache) and extracted a tidy map.
- You persisted **provenance** and used it to enrich your dataset.
- Keep scrapers **small, cached, and resilient**.

---

## 14.5 Homework (due next week)

**Goal:** Document your web data provenance and generate a minimal **data dictionary** for the project.

### 14.5.1 A. Provenance section (script that composes a Markdown file)

```
# scripts/write_provenance.py
#!/usr/bin/env python
import json, pandas as pd
from pathlib import Path
Path("reports").mkdir(exist_ok=True)

provenance = []
if Path("data/static/sector_map.provenance.json").exists():
    provenance.append(json.loads(Path("data/static/sector_map.provenance.json").read_text())
else:
    provenance.append({"source_url":"(none)","fetched_at_utc":"(n/a)"})

md = ["# Data provenance",
      "",
      "## Web sources",
      "",
      "| Source | Fetched at |",
```

```
        "|---|---|"]
for p in provenance:
    md.append(f"| {p['source_url']} | {p['fetched_at_utc']} |")

Path("reports/provenance.md").write_text("\n".join(md))
print("Wrote reports/provenance.md")
```

Run:

```
%%bash
chmod +x scripts/write_provenance.py
python scripts/write_provenance.py
```

### 14.5.2 B. Data dictionary generator

```
# scripts/data_dictionary.py
#!/usr/bin/env python
import pandas as pd
from pathlib import Path

def describe_parquet(path):
    df = pd.read_parquet(path)
    dtypes = df.dtypes.astype(str).to_dict()
    return pd.DataFrame({"column": list(dtypes.keys()), "dtype": list(dtypes.values())})

def main():
    rows=[]
    for path in ["data/processed/prices.parquet",
                 "data/processed/returns.parquet",
                 "data/processed/features_v1.parquet",
                 "data/processed/features_v1_ext.parquet"]:
        p = Path(path)
        if p.exists():
            df = describe_parquet(p)
            df.insert(0, "dataset", p.name)
            rows.append(df)
    out = pd.concat(rows, ignore_index=True) if rows else pd.DataFrame(columns=["dataset","co
    Path("reports").mkdir(exist_ok=True)
    out.to_csv("reports/data_dictionary.csv", index=False)
    print("Wrote reports/data_dictionary.csv")
```

```python
if __name__ == "__main__":
    main()
```

Run:

```bash
%%bash
chmod +x scripts/data_dictionary.py
python scripts/data_dictionary.py
```

### 14.5.3 C. (Optional) Add a short Quarto page that includes both files

Create `reports/data_overview.qmd` and render in your next report.

### 14.5.4 D. Quick tests

```python
# tests/test_dictionary_provenance.py
import os, pandas as pd
def test_provenance_and_dict():
    assert os.path.exists("reports/provenance.md")
    assert os.path.exists("reports/data_dictionary.csv")
    df = pd.read_csv("reports/data_dictionary.csv")
    assert {"dataset","column","dtype"}.issubset(df.columns)
```

Run:

```bash
%%bash
pytest -q
```

---

## 14.6 Instructor tips (for all three sessions)

- Keep a one-page "**no leakage**" checklist handy and point to it often.
- For Session 11, have a prepared `.env` with a working FRED key to avoid classroom delays.

- For Session 12, if Wikipedia blocks requests, switch to `pandas.read_html` (shown) or use a small pre-saved HTML in `data/static/` to demonstrate parsing.

These three sessions carry you from solid **feature engineering** → **external data integration** → **web scraping with ethics**, setting up a strong foundation for the testing/CI weeks that follow.

# 15 Session 13

Below is a complete lecture package for **Session 13 — pytest + Data Validation** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. You'll add **high-value tests** around your features and a **Pandera** (optional) schema, practice **logging**, and wire everything so tests run fast and deterministically.

> **Assumptions:** You completed Session 9–12 and have `data/processed/features_v1.parquet` (or `features_v1_ext.parquet`). If a file is missing, the lab provides a small synthetic fallback so tests still run. **Goal today:** Make it **hard to ship bad data** by adding precise, fast tests.

---

## 15.1 Session 13 — pytest + Data Validation (75 min)

### 15.1.1 Learning goals

By the end of class, students can:

1. Write **fast, high-signal tests** for data pipelines (shapes, dtypes, nulls, **no look-ahead**).
2. Validate a DataFrame with **Pandera** (schema + value checks) or **custom checks** only.
3. Use **logging** effectively and capture logs in tests.
4. Run tests in Colab / locally and prepare for CI in Session 14.

---

## 15.2 Agenda (75 min)

- **(10 min)** Slides: What to test (and not), "data tests" vs unit tests, speed budget
- **(10 min)** Slides: Pandera schemas & custom checks; tolerance and stability
- **(10 min)** Slides: Logging basics (`logging`, levels, handlers); testing logs with `caplog`

- **(35 min) In-class lab**: add `tests/test_features.py` (+ optional Pandera test), fixtures, config; run & fix
- **(10 min)** Wrap-up + homework briefing

---

## 15.3 Slides / talking points (drop into your deck)

### 15.3.1 What to test (fast, crisp)

- **Contract tests** for data:
  - **Schema**: required columns exist; dtypes sane (`ticker` categorical, calendar ints).
  - **Nulls**: no NAs in training-critical cols.
  - **Semantics**: `r_1d` is **lead** of `log_return`; rolling features computed from **past only**.
  - **Keys**: no duplicate `(ticker, date)`; dates strictly increasing within ticker.
- Keep tests **under ~5s total** (CI budget). Avoid long recomputations; sample/take head.

### 15.3.2 Pandera vs custom checks

- **Pandera**: declarative schema; optional dependency; good for **column existence + ranges**.
- **Custom**: essential for **domain logic** (look-ahead bans, exact rolling formulas).

### 15.3.3 Logging basics

- Use `logging.getLogger(__name__)`; set level via env (`LOGLEVEL=INFO`).
- Log **counts, ranges, and any data drops** inside build scripts.
- In tests: use `caplog` to assert a warning is emitted for suspicious conditions.

---

## 15.4 In-class lab (35 min)

Run each block as its **own Colab cell**. Adjust `REPO_NAME` as needed.

### 15.4.1 0) Setup: mount & folders

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"  # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in ["data/processed","tests","scripts","reports"]:
    pathlib.Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())
```

### 15.4.2 1) (Optional) Install test-time helpers (Pandera)

```
!pip -q install pytest pandera pyarrow
```

### 15.4.3 2) Put a tiny logging helper in your repo (used by build scripts & tests)

```
# scripts/logsetup.py
from __future__ import annotations
import logging, os

def setup_logging(name: str = "dspt"):
    level = os.getenv("LOGLEVEL", "INFO").upper()
    logger = logging.getLogger(name)
    if not logger.handlers:
        handler = logging.StreamHandler()
        fmt = "%(asctime)s | %(levelname)s | %(name)s | %(message)s"
        handler.setFormatter(logging.Formatter(fmt))
        logger.addHandler(handler)
    logger.setLevel(level)
    return logger
```

### 15.4.4 3) Create pytest config and a fixture (with safe fallback data)

```
# pytest.ini
from pathlib import Path
Path("pytest.ini").write_text("""[pytest]
addopts = -q
testpaths = tests
filterwarnings =
    ignore::FutureWarning
""")

# tests/conftest.py
from pathlib import Path
import pandas as pd, numpy as np, pytest

def _synth_features():
    # minimal synthetic features for 3 tickers, 60 days
    rng = np.random.default_rng(0)
    dates = pd.bdate_range("2023-01-02", periods=60)
    frames=[]
    for t in ["AAPL","MSFT","GOOGL"]:
        ret = rng.normal(0, 0.01, size=len(dates)).astype("float32")
        adj = 100 * np.exp(np.cumsum(ret))
        df = pd.DataFrame({
            "date": dates,
            "ticker": t,
            "adj_close": adj.astype("float32"),
            "log_return": np.r_[np.nan, np.diff(np.log(adj))].astype("float32")
        })
        # next-day label
        df["r_1d"] = df["log_return"].shift(-1)
        # rolling
        df["roll_mean_20"] = df["log_return"].rolling(20, min_periods=20).mean()
        df["roll_std_20"]  = df["log_return"].rolling(20, min_periods=20).std()
        df["zscore_20"]    = (df["log_return"]-df["roll_mean_20"])/(df["roll_std_20"]+1e-8)
        df["weekday"] = df["date"].dt.weekday.astype("int8")
        df["month"]   = df["date"].dt.month.astype("int8")
        frames.append(df)
    out = pd.concat(frames, ignore_index=True).dropna().reset_index(drop=True)
    out["ticker"] = out["ticker"].astype("category")
    return out
```

```python
@pytest.fixture(scope="session")
def features_df():
    p = Path("data/processed/features_v1.parquet")
    if p.exists():
        df = pd.read_parquet(p)
        # Ensure expected minimal cols exist (compute light ones if missing)
        if "weekday" not in df: df["weekday"] = pd.to_datetime(df["date"]).dt.weekday.astype
        if "month" not in df:   df["month"] = pd.to_datetime(df["date"]).dt.month.astype("in
        return df.sort_values(["ticker","date"]).reset_index(drop=True)
    # fallback
    return _synth_features().sort_values(["ticker","date"]).reset_index(drop=True)
```

### 15.4.5 4) High-value tests: shapes, nulls, look-ahead ban (as requested)

```python
# tests/test_features.py
import numpy as np, pandas as pd
import pytest

REQUIRED_COLS = ["date","ticker","log_return","r_1d","weekday","month"]

def test_required_columns_present(features_df):
    missing = [c for c in REQUIRED_COLS if c not in features_df.columns]
    assert not missing, f"Missing required columns: {missing}"

def test_key_no_duplicates(features_df):
    dup = features_df[["ticker","date"]].duplicated().sum()
    assert dup == 0, f"Found {dup} duplicate (ticker,date) rows"

def test_sorted_within_ticker(features_df):
    for tkr, g in features_df.groupby("ticker"):
        assert g["date"].is_monotonic_increasing, f"Dates not sorted for {tkr}"

def test_nulls_in_critical_columns(features_df):
    crit = ["log_return","r_1d"]
    na = features_df[crit].isna().sum().to_dict()
    assert all(v == 0 for v in na.values()), f"NAs in critical cols: {na}"

def test_calendar_dtypes(features_df):
    assert str(features_df["weekday"].dtype) in ("int8","Int8"), "weekday should be compact i
    assert str(features_df["month"].dtype)   in ("int8","Int8"), "month should be compact in
```

```
def test_ticker_is_categorical(features_df):
    # allow object if reading from some parquet engines, but prefer category
    assert features_df["ticker"].dtype.name in ("category","CategoricalDtype","object")

def test_r1d_is_lead_of_log_return(features_df):
    for tkr, g in features_df.groupby("ticker"):
        # r_1d at t equals log_return at t+1
        assert g["r_1d"].iloc[:-1].equals(g["log_return"].iloc[1:]), f"Lead/lag mismatch for

@pytest.mark.parametrize("W", [20])
def test_rolling_mean_matches_definition(features_df, W):
    if f"roll_mean_{W}" not in features_df.columns:
        pytest.skip(f"roll_mean_{W} not present")
    for tkr, g in features_df.groupby("ticker"):
        s = g["log_return"]
        rm = s.rolling(W, min_periods=W).mean()
        # compare only where defined
        mask = ~rm.isna()
        diff = (g[f"roll_mean_{W}"][mask] - rm[mask]).abs().max()
        assert float(diff) <= 1e-7, f"roll_mean_{W} mismatch for {tkr} (max diff {diff})"
```

### 15.4.6 5) Optional Pandera schema test (declarative)

```
# tests/test_schema_pandera.py
import pytest, pandas as pd, numpy as np
try:
    import pandera as pa
    from pandera import Column, Check, DataFrameSchema
except Exception:
    pytest.skip("pandera not installed", allow_module_level=True)

schema = pa.DataFrameSchema({
    "date":    Column(pa.DateTime, nullable=False),
    "ticker":  Column(pa.String, nullable=False, coerce=True, checks=Check.str_length(1, 12)
    "log_return": Column(pa.Float, nullable=False, checks=Check.is_finite()),
    "r_1d":       Column(pa.Float, nullable=False, checks=Check.is_finite()),
    "weekday":    Column(pa.Int8,  checks=Check.in_range(0, 6)),
    "month":      Column(pa.Int8,  checks=Check.in_range(1, 12)),
}, coerce=True, strict=False)
```

```python
def test_schema_validate(features_df):
    # Cast ticker to string for schema validation; categorical is ok → string
    df = features_df.copy()
    df["ticker"] = df["ticker"].astype(str)
    schema.validate(df[["date","ticker","log_return","r_1d","weekday","month"]])
```

### 15.4.7 6) Logging test: assert a warning is emitted on duplicates (toy demo)

```python
# tests/test_logging.py
import logging, pandas as pd, numpy as np, pytest
from scripts.logsetup import setup_logging

def check_for_duplicates(df, logger=None):
    logger = logger or setup_logging("dspt")
    dups = df[["ticker","date"]].duplicated().sum()
    if dups > 0:
        logger.warning("Found %d duplicate (ticker,date) rows", dups)
    return dups

def test_duplicate_warning(caplog):
    caplog.set_level(logging.WARNING)
    df = pd.DataFrame({"ticker":["AAPL","AAPL"], "date":pd.to_datetime(["2024-01-02","2024-0
    dups = check_for_duplicates(df)
    assert dups == 1
    assert any("duplicate" in rec.message for rec in caplog.records)
```

### 15.4.8 7) Run tests now

```
!pytest -q
```

If a test fails on your real data, fix your pipeline (e.g., regenerate `features_v1.parquet`) and re-run. **Do not** relax the test without understanding the failure.

---

## 15.5 Wrap-up (10 min)

- You now have **tests that fail loudly** if labels leak, required columns/keys break, or schemas drift.
- Pandera provides a declarative baseline; custom tests encode your **domain logic**.
- Logging helps you **debug data issues**; you can assert on log messages in tests.

---

## 15.6 Homework (due before Session 14)

**Goal:** Create a **Health Check** notebook that prints key diagnostics and is easy to include in your Quarto report.

### 15.6.1 Part A — Build a reusable health module

```python
# scripts/health.py
from __future__ import annotations
import pandas as pd, numpy as np, json
from pathlib import Path

def df_health(df: pd.DataFrame) -> dict:
    out = {}
    out["rows"] = int(len(df))
    out["cols"] = int(df.shape[1])
    out["date_min"] = str(pd.to_datetime(df["date"]).min().date())
    out["date_max"] = str(pd.to_datetime(df["date"]).max().date())
    out["tickers"]  = int(df["ticker"].nunique())
    # Null counts (top 10)
    na = df.isna().sum().sort_values(ascending=False)
    out["nulls"] = na[na>0].head(10).to_dict()
    # Duplicates
    out["dup_key_rows"] = int(df[["ticker","date"]].duplicated().sum())
    # Example numeric ranges for core cols
    for c in [x for x in ["log_return","r_1d","roll_std_20"] if x in df.columns]:
        s = pd.to_numeric(df[c], errors="coerce")
        out[f"{c}_min"] = float(np.nanmin(s))
        out[f"{c}_max"] = float(np.nanmax(s))
    return out
```

```python
def write_health_report(in_parquet="data/processed/features_v1.parquet",
                        out_json="reports/health.json", out_md="reports/health.md"):
    p = Path(in_parquet)
    if not p.exists():
        raise SystemExit(f"Missing {in_parquet}.")
    df = pd.read_parquet(p)
    h = df_health(df)
    Path(out_json).write_text(json.dumps(h, indent=2))
    # Render a small Markdown summary
    lines = [
        "# Data Health Summary",
        "",
        f"- Rows: **{h['rows']}**; Cols: **{h['cols']}**; Tickers: **{h['tickers']}**",
        f"- Date range: **{h['date_min']} → {h['date_max']}**",
        f"- Duplicate (ticker,date) rows: **{h['dup_key_rows']}**",
    ]
    if h.get("nulls"):
        lines += ["", "## Top Null Counts", ""]
        lines += [f"- **{k}**: {v}" for k,v in h["nulls"].items()]
    Path(out_md).write_text("\n".join(lines))
    print("Wrote", out_json, "and", out_md)
```

Run once to generate the files:

```
!python scripts/health.py
```

### 15.6.2 Part B — Health Check notebook (`reports/health.ipynb`)

Create a new notebook `reports/health.ipynb` with **two cells**:

**Cell 1 (setup):**

```python
%load_ext autoreload
%autoreload 2
from scripts.health import write_health_report
write_health_report()  # writes reports/health.json and reports/health.md
```

**Cell 2 (display in notebook):**

```python
from pathlib import Path
print(Path("reports/health.md").read_text())
```

Commit the notebook. It will be light and re-usable. You'll include its output in Quarto below.

### 15.6.3 Part C — Include health output in your Quarto report

In `reports/eda.qmd`, add a section:

```
## Data Health (auto-generated)

::: {.cell execution_count=1}
````` {.python .cell-code}
from pathlib import Path
print(Path("reports/health.md").read_text())
```

```
Render EDA:
```bash
quarto render reports/eda.qmd
```

### 15.6.4 Part D — Add a Makefile target and a quick test

**Makefile append:**

```
.PHONY: health test
health: ## Generate health.json and health.md from the current features parquet
\tpython scripts/health.py

test: ## Run fast tests
\tpytest -q
```

:::

**Test that health files exist:**

```python
# tests/test_health_outputs.py
import os, json

def test_health_files_exist():
    assert os.path.exists("reports/health.json")
    assert os.path.exists("reports/health.md")
    # json is valid
    import json
    json.load(open("reports/health.json"))
```

Run:

```bash
%%bash
make health
pytest -q -k health
```

---

## 15.7 Instructor checklist (before class)

- Ensure `features_v1.parquet` exists or the fixture's synthetic fallback works.
- Dry-run `pytest -q` in a fresh runtime; keep total time $< 5$s.
- Prepare 2–3 "expected failures" you can toggle (e.g., edit one feature column to NaN) to show tests catching bugs.

## 15.8 Emphasize while teaching

- **Fast tests only** for CI; keep heavy, long recomputations out.
- **No look-ahead** and **unique (ticker,date)** are non-negotiable contracts.
- Logging is a first-class tool—tests can assert on **warnings** you emit.

## 15.9 Grading (pass/revise)

- `tests/test_features.py` present with **shapes, nulls, look-ahead ban** (and rolling check).
- Tests pass locally (`pytest -q`).

- `reports/health.ipynb` and `reports/health.md/.json` exist and integrate into `eda.qmd`.
- Makefile `health` and `test` targets work.

You now have a **safety net** around your data. In **Session 14**, we'll enforce style with **pre-commit** and bring your tests to **GitHub Actions CI**.

# 16 pre-commit & GitHub Actions CI

Below is a complete lecture package for **Session 14 — pre-commit & GitHub Actions CI** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. By the end, your repo will (1) enforce **style and lint** automatically with **pre-commit** (Black, Ruff, nbstripout), and (2) run **CI** on every PR with a fast GitHub Actions workflow that lints and runs tests in under ~3–4 minutes.

> **Assumptions:** You completed Session 13 and have a repo in Drive (e.g., `unified-stocks-teamX`) with a small test suite (`pytest`) and Parquet data present locally. Colab + Drive workflow assumed. **Goals today:** Make code quality and basic data tests automatic and repeatable in CI.

---

## 16.1 Session 14 — pre-commit & GitHub Actions CI (75 min)

### 16.1.1 Learning goals

Students will be able to:

1. Configure **pre-commit** to run **Black**, **Ruff** (lint + import sort), and **nbstripout** on every commit.
2. Keep commits clean and **notebook outputs stripped**.
3. Add a fast **GitHub Actions** CI workflow that runs pre-commit hooks and **pytest** on each PR.
4. Keep CI runtime **under ~3–4 minutes** with caching and a lean dependency set.

---

## 16.2 Agenda (75 min)

- **(10 min)** Slides: why pre-commit; the "quality gate"; anatomy of a fast CI
- **(10 min)** Slides: Black vs Ruff; when nbstripout matters; what belongs in CI
- **(35 min) In-class lab**: configure pre-commit (Black, Ruff, nbstripout) → run locally → add CI workflow → local dry-run
- **(10 min)** Wrap-up + homework briefing
- **(10 min)** Buffer

---

## 16.3 Main points

**Why pre-commit?**

- Prevent "drive-by" problems before they enter history: unformatted code, stray notebook outputs, trailing whitespace.
- Hooks run **locally on commit**, then again in **CI** for defense-in-depth.

**Black & Ruff**

- **Black**: opinionated formatter → consistent diffs; no bikeshedding.
- **Ruff**: very fast linter (flake8 family), plus **import sorting**; can also fix many issues (`--fix`).
- You can use **both** (common) or let Ruff handle formatting too; we'll use both for clarity.

**nbstripout**

- Remove cell outputs from notebooks to keep diffs small, avoid binary bloat, and reduce CI time.
- Two patterns: **pre-commit hook** (recommended) and/or **git filter** (`nbstripout --install`).

**CI scope (fast!)**

- Lint + tests only; **no heavy training** in CI.
- Cache dependencies; pin Python (3.11+).
- Keep tests deterministic and **< ~5s** (already done in Session 13).

---

## 16.4 In-class lab (35 min, Colab-friendly)

Run each block as its **own Colab cell**. Update `REPO_NAME` to your repo. The cells create and modify files inside your repo.

### 16.4.1 0) Mount Drive & go to repo

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"  # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in [".github/workflows","tests","scripts","reports"]:
    pathlib.Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())
```

### 16.4.2 1) Install tools locally (for this Colab runtime)

```
!pip -q install pre-commit black ruff nbstripout pytest
```

### 16.4.3 2) Add tool config to `pyproject.toml` (Black + Ruff)

If you don't have a `pyproject.toml`, this cell will create a minimal one; otherwise it appends/updates sections.

```
from pathlib import Path
import textwrap, re

pyproj = Path("pyproject.toml")
existing = pyproj.read_text() if pyproj.exists() else ""

def upsert(section_header, body):
    global existing
```

```
    pattern = rf"(?ms)^\[{re.escape(section_header)}\]\s*.*?(?=^\[|\Z)"
    if re.search(pattern, existing):
        existing = re.sub(pattern, f"[{section_header}]\n{body}\n", existing)
    else:
        existing += f"\n[{section_header}]\n{body}\n"

# Black
upsert("tool.black", textwrap.dedent("""
line-length = 88
target-version = ["py311"]
""").strip())

# Ruff (modern layout)
upsert("tool.ruff", textwrap.dedent("""
line-length = 88
target-version = "py311"
""").strip())

upsert("tool.ruff.lint", textwrap.dedent("""
select = ["E","F","I"]  # flake8 errors, pyflakes, import sort
ignore = ["E501"]       # let Black handle line length
""").strip())

upsert("tool.ruff.lint.isort", textwrap.dedent("""
known-first-party = ["projectname"]
""").strip())

pyproj.write_text(existing.strip()+"\n")
print(pyproj.read_text())
```

### 16.4.4 3) Create `.pre-commit-config.yaml` with hooks (Black, Ruff, nbstripout)

Versions below are stable at time of writing—feel free to bump later.

```
from pathlib import Path
cfg = Path(".pre-commit-config.yaml")
cfg.write_text("""repos:
  - repo: https://github.com/psf/black
    rev: 24.4.2
    hooks:
      - id: black
```

```
        language_version: python3.11

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.5.0
    hooks:
      - id: ruff
        args: [--fix, --exit-non-zero-on-fix]
      - id: ruff-format

  - repo: https://github.com/kynan/nbstripout
    rev: 0.7.1
    hooks:
      - id: nbstripout
        files: \\.ipynb$

  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.6.0
    hooks:
      - id: end-of-file-fixer
      - id: trailing-whitespace
      - id: check-yaml
      - id: check-added-large-files
""")
print(cfg.read_text())
```

### 16.4.5 4) Install the local git hook & run on all files

```
!pre-commit install
!pre-commit run --all-files
```

> The first run will **download** hook toolchains (Black, Ruff, etc.), format files, and strip notebook outputs. Commit changes after verifying.

### 16.4.6 5) (Optional) Also install git filter for nbstripout

> This is an extra layer; pre-commit hook above already strips outputs. Use this to guarantee outputs are removed even when bypassing pre-commit.

```
!nbstripout --install --attributes .gitattributes
print(open(".gitattributes").read())
```

#### 16.4.7 6) Add a tiny "bad style" file to see hooks in action

```
from pathlib import Path
p = Path("scripts/bad_style.py")
p.write_text("import os,sys\n\n\ndef add(a,b):\n  return(a +  b)\n")
print("Wrote:", p)

# Run hooks just on this file
!pre-commit run --files scripts/bad_style.py
print(open("scripts/bad_style.py").read())
```

You should see Black and Ruff fix spacing/imports; trailing whitespace hooks may also fire.

#### 16.4.8 7) Add a fast GitHub Actions CI workflow (`.github/workflows/ci.yml`)

This runs pre-commit and your tests on Ubuntu with Python 3.11, with pip caching.

```
from pathlib import Path
wf = Path(".github/workflows/ci.yml")
wf.write_text("""name: CI
on:
  push:
    branches: [ main, master, develop ]
  pull_request:
    branches: [ main, master, develop ]
concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true
jobs:
  build:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v4
```

```
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
          cache: 'pip'
          cache-dependency-path: |
            requirements.txt
            pyproject.toml

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
          pip install pre-commit pytest

      # Run pre-commit (Black, Ruff, nbstripout, etc.)
      - name: pre-commit
        uses: pre-commit/action@v3.0.1

      # Run tests (fast only)
      - name: pytest
        run: pytest -q --maxfail=1
""")
print(wf.read_text())
```

### 16.4.9 8) Add a `Makefile` convenience (optional but nice)

```
from pathlib import Path
mk = Path("Makefile")
text = mk.read_text() if mk.exists() else ""
if "lint" not in text:
    text += """

.PHONY: lint test ci-local
lint: ## Run pre-commit hooks on all files
\tpre-commit run --all-files

test: ## Run fast tests
\tpytest -q --maxfail=1

ci-local: lint test ## Simulate CI locally
```

```
"""
    mk.write_text(text)
print(mk.read_text())
```

---

## 16.5 Wrap-up (10 min)

- You configured **pre-commit** with **Black**, **Ruff** (lint + import sort), and **nbstripout** to keep the repo clean.
- You added a fast **CI** that runs the same hooks plus **pytest** on every PR.
- CI time stays small due to **caching** and a **lean dependency set**; tests are fast by design (Session 13).

---

## 16.6 Homework (due before next session)

**Goal:** Prove the workflow works end-to-end with a green PR from a **fresh clone**.

### 16.6.1 Part A — Fresh-clone smoke test (local)

```
# On your laptop or a new Colab session:
git clone https://github.com/YOUR_USER/unified-stocks-teamX.git
cd unified-stocks-teamX
python -m pip install -U pip
pip install pre-commit pytest
pre-commit install
pre-commit run --all-files
pytest -q --maxfail=1
```

### 16.6.2 Part B — Open a PR that turns CI green

1. **Create a branch** and make a tiny, style-breaking change, then commit and let pre-commit fix it automatically.

```
git checkout -b chore/ci-badge-and-hooks
echo "# Tiny edit  " >> README.md   # trailing spaces (will be fixed)
git add -A
git commit -m "chore: add CI badge + enable pre-commit hooks"
git push -u origin chore/ci-badge-and-hooks
```

2. **Add a CI badge** to `README.md`:

```
![CI](https://github.com/YOUR_USER/unified-stocks-teamX/actions/workflows/ci.yml/badge.s
```

3. Open a **Pull Request** on GitHub. Verify that:

   - The **pre-commit** step passes.
   - **pytest** passes.
   - Total runtime is $<$ ~**3–4 minutes**.

4. Merge once green. (If red, fix locally; do *not* disable hooks.)

### 16.6.3 Part C — (Optional) Tune Ruff + Black to your taste

- In `pyproject.toml`, try:

```
[tool.black]
line-length = 100

[tool.ruff]
line-length = 100

[tool.ruff.lint]
select = ["E","F","I","B"]  # enable flake8-bugbear
ignore = ["E501"]
```

- Run `pre-commit run --all-files` and ensure CI remains green.

### 16.6.4 Part D — (Optional) Add notebook QA without executing them

- Add **nbqa** to run Ruff on notebooks (markdown & code cells):

```
# append to .pre-commit-config.yaml
- repo: https://github.com/nbQA-dev/nbQA
  rev: 1.8.5
  hooks:
    - id: nbqa-ruff
```

```yaml
      args: [--fix]
      additional_dependencies: [ruff==0.5.0]
```

- Re-install hooks and run `pre-commit run --all-files`.

---

## 16.7 Reference checklist (for grading)

- `.pre-commit-config.yaml` present with **Black, Ruff, nbstripout**.
- `pyproject.toml` includes [**tool.black**] and [**tool.ruff**] sections.
- `.github/workflows/ci.yml` runs **pre-commit** and **pytest** with **Python 3.11** and pip caching.
- `make lint`, `make test`, `make ci-local` work (if you added them).
- A PR was opened and CI is **green**; README has the **CI badge**.

---

## 16.8 Instructor tips / gotchas

- If pre-commit says "no files to check" for nbstripout, ensure your **file matcher** `files: \.ipynb$` is correct and that notebooks are tracked.
- If Ruff conflicts with Black on formatting: keep **Black** as the authority, disable `E501` in Ruff, and let Ruff handle **imports** (`I`) and errors (`E`, `F`).
- CI failures from missing deps: ensure your `requirements.txt` (or `pyproject.toml` with `[project.dependencies]`) includes **pandas**, **pyarrow**, and **pytest** if your tests read Parquet.
- Keep CI lean: no data downloads or training; use **fixtures** and tiny synthetic datasets (Session 13 pattern).

You now have an automated quality gate—**style, lint, and tests run locally and in CI**—so your future PRs start green and stay green.

# 17 Session 15 — Framing & Metrics (Rolling-Origin Evaluation)

Below is a complete lecture package for **Session 15 — Framing & Metrics (Rolling-Origin Evaluation)** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. You'll formalize the forecasting problem (horizon/step), implement a **rolling-origin splitter** (a.k.a. walk-forward), and evaluate **naive** and **seasonal-naive** baselines with **MAE, sMAPE, MASE**, aggregated **across tickers** (macro vs micro/weighted).

> **Educational use only — not trading advice.** Assumes your repo in Drive (e.g., `unified-stocks-teamX`) and `data/processed/returns.parquet` from Session 9. If missing, the lab creates a small fallback.

---

## 17.1 Session 15 — Framing & Metrics (75 min)

### 17.1.1 Learning goals

By the end of class, students can:

1. Specify **forecast horizon** $H$, **step** (stride), and choose between **expanding** vs **sliding** rolling-origin evaluation with an **embargo** gap.
2. Implement a **date-based splitter** that yields `(train_idx, val_idx)` for all tickers at once.
3. Compute **MAE**, **sMAPE**, **MASE** (with a proper **training-window scale**), and aggregate **per-ticker** and **across tickers** (macro vs micro/weighted).
4. Produce a tidy CSV of baseline results to serve as your course's ground truth.

---

## 17.2 Agenda (75 min)

- **(10 min)** Slides: forecasting setup — horizon $H$, step, rolling-origin (expanding vs sliding), embargo
- **(10 min)** Slides: metrics — MAE, sMAPE, MASE; aggregation across tickers (macro vs micro/weighted)
- **(35 min) In-class lab**: implement a date-based splitter → compute naive & seasonal-naive baselines → MAE/sMAPE/MASE per split/ticker → save reports
- **(10 min)** Wrap-up & homework brief
- **(10 min)** Buffer

---

## 17.3 Slides / talking points (add these bullets to your deck)

### 17.3.1 Framing the forecast

- **Target:** next-day log return $r_{t+1}$ (you built this as `r_1d`).
- **Horizon $H$:** 1 business day.
- **Step (stride):** how far the **origin** moves forward each split (e.g., 63 trading days a quarter).
- **Rolling-origin schemes**
  - **Expanding:** train start fixed; **train grows** over time.
  - **Sliding (rolling):** fixed-length train window **slides** forward.
- **Embargo:** small **gap** (e.g., 5 days) between train end and validation start to avoid adjacency leakage.

### 17.3.2 Metrics (scalar, easy to compare)

- **MAE:** $\frac{1}{n}\sum |y-\hat{y}|$ — robust & interpretable.
- **sMAPE:** $\frac{2}{n}\sum \frac{|y-\hat{y}|}{(|y|+|\hat{y}|+\epsilon)}$ — scale-free, safe for near-zero returns with $\epsilon$.
- **MASE:** $\text{MASE} = \frac{\text{MAE}_{\text{model}}}{\text{MAE}_{\text{naive (train)}}}$ — $<1$ means better than naive.
  - For seasonality $s$, the **naive comparator** predicts $y_{t+1} \approx y_{t+1-s}$ (we'll use $s = 5$ for day-of-week seasonality on business days).
  - **Scale** is computed on the **training window only**, per ticker.

### 17.3.3 Aggregation across tickers

- **Per-ticker metrics** first → then aggregate.
- **Macro average:** mean of per-ticker metrics (each ticker equal weight).
- **Micro/weighted:** pool all rows (or weight tickers by sample count); for MAE, pooled MAE equals sample-count weighted average of per-ticker MAEs.

---

## 17.4 In-class lab (35 min, Colab-friendly)

Run each block as its own cell. Adjust `REPO_NAME` if needed.

### 17.4.1 0) Setup & fallback data

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"  # <- change to your repo name
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, numpy as np, pandas as pd
from pathlib import Path
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in ["data/raw","data/processed","reports","scripts","tests"]:
    Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())

# Load returns or create a tiny fallback
rpath = Path("data/processed/returns.parquet")
if rpath.exists():
    returns = pd.read_parquet(rpath)
else:
    # Fallback synthetic returns for 5 tickers, 320 business days
    rng = np.random.default_rng(0)
    dates = pd.bdate_range("2022-01-03", periods=320)
    frames=[]
```

```
    for tkr in ["AAPL","MSFT","GOOGL","AMZN","NVDA"]:
        eps = rng.normal(0, 0.012, size=len(dates)).astype("float32")
        adj = 100*np.exp(np.cumsum(eps))
        df = pd.DataFrame({
            "date": dates,
            "ticker": tkr,
            "adj_close": adj.astype("float32"),
            "log_return": np.r_[np.nan, np.diff(np.log(adj))].astype("float32")
        })
        df["r_1d"] = df["log_return"].shift(-1)
        df["weekday"] = df["date"].dt.weekday.astype("int8")
        df["month"]   = df["date"].dt.month.astype("int8")
        frames.append(df)
    returns = pd.concat(frames, ignore_index=True).dropna().reset_index(drop=True)
    returns["ticker"] = returns["ticker"].astype("category")
    returns.to_parquet(rpath, index=False)

# Standardize
returns["date"] = pd.to_datetime(returns["date"])
returns = returns.sort_values(["ticker","date"]).reset_index(drop=True)
returns["ticker"] = returns["ticker"].astype("category")
returns.head()
```

### 17.4.2 1) Rolling-origin date splitter (expanding windows + embargo)

```
import numpy as np, pandas as pd

def make_rolling_origin_splits(dates: pd.Series,
                               train_min=252,   # ~1y of trading days
                               val_size=63,     # ~1 quarter
                               step=63,
                               embargo=5):
    """Return a list of (train_start, train_end, val_start, val_end) date tuples."""
    u = np.array(sorted(pd.to_datetime(dates.unique())))
    n = len(u)
    splits=[]
    i = train_min - 1
    while True:
        if i >= n: break
        tr_start, tr_end = u[0], u[i]
```

```
        vs_idx = i + embargo + 1
        ve_idx = vs_idx + val_size - 1
        if ve_idx >= n: break
        splits.append((tr_start, tr_end, u[vs_idx], u[ve_idx]))
        i += step
    return splits

def splits_to_indices(df, split):
    """Map a date split to index arrays for the full multi-ticker frame."""
    a,b,c,d = split
    tr_idx = df.index[(df["date"]>=a) & (df["date"]<=b)].to_numpy()
    va_idx = df.index[(df["date"]>=c) & (df["date"]<=d)].to_numpy()
    # sanity: embargo => last train date < first val date
    assert b < c
    return tr_idx, va_idx

splits = make_rolling_origin_splits(returns["date"], train_min=252, val_size=63, step=63, emb
len(splits), splits[:2]
```

### 17.4.3 2) Metrics & baseline predictors (naive and seasonal-naive)

```
from typing import Dict, Tuple

def mae(y, yhat):
    y = np.asarray(y); yhat = np.asarray(yhat);
    return float(np.mean(np.abs(y - yhat)))

def smape(y, yhat, eps=1e-8):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(2.0*np.abs(y - yhat)/(np.abs(y)+np.abs(yhat)+eps)))

def mase(y_true, y_pred, y_train_true, y_train_naive):
    # Scale = MAE of comparator (naive) on TRAIN only; add tiny epsilon
    scale = mae(y_train_true, y_train_naive) + 1e-12
    return float(mae(y_true, y_pred) / scale)

def add_baseline_preds(df: pd.DataFrame, seasonality:int=5) -> pd.DataFrame:
    """
    For each ticker:
      - naive predicts r_{t+1}   log_return_t (s=1)
```

```
      - seasonal naive (s) predicts r_{t+1}   log_return_{t+1-s}  => shift(s-1)
    Adds columns: yhat_naive, yhat_s{s}
    """
    out = df.copy()
    out["yhat_naive"] = out.groupby("ticker")["log_return"].transform(lambda s: s)  # s=1
    if seasonality <= 1:
        out["yhat_s"] = out["yhat_naive"]
    else:
        out["yhat_s"] = out.groupby("ticker")["log_return"].transform(lambda s: s.shift(seas
    return out
```

#### 17.4.4 3) Evaluate baselines across first 2 splits (fast in class)

```
# Precompute predictions over the entire frame once (safe: uses only past values via shift)
seasonality = 5  # business-day weekly
preds_all = add_baseline_preds(returns, seasonality=seasonality)

def per_ticker_metrics(df_val, df_train, method="naive") -> pd.DataFrame:
    """
    Compute per-ticker MAE, sMAPE, MASE for the chosen method ('naive' or 's').
    MASE scale uses TRAIN window and the same comparator as method.
    """
    rows=[]
    col = "yhat_naive" if method=="naive" else "yhat_s"
    for tkr, g in df_val.groupby("ticker"):
        gv = g.dropna(subset=["r_1d", col])
        if len(gv)==0:
            continue
        # TRAIN scale (per ticker)
        gt = df_train[df_train["ticker"]==tkr].dropna(subset=["r_1d"])
        if method=="naive":
            gt_pred = gt["log_return"]  # s=1
        else:
            gt_pred = gt["log_return"].shift(seasonality-1)
        gt_clean = gt.dropna(subset=["r_1d"]).copy()
        gt_pred = gt_pred.loc[gt_clean.index]
        gt_clean = gt_clean.dropna(subset=["r_1d"])
        # Align indices
        y_tr = gt_clean["r_1d"].to_numpy()
        yhat_tr_naive = gt_pred.to_numpy()
```

```
        # VAL metrics
        y = gv["r_1d"].to_numpy()
        yhat = gv[col].to_numpy()
        rows.append({
            "ticker": tkr,
            "n": int(len(y)),
            "mae": mae(y,yhat),
            "smape": smape(y,yhat),
            "mase": mase(y, yhat, y_tr, yhat_tr_naive),
        })
    return pd.DataFrame(rows)

def aggregate_across_tickers(per_ticker_df: pd.DataFrame) -> Dict[str,float]:
    if per_ticker_df.empty:
        return {"macro_mae":np.nan,"macro_smape":np.nan,"macro_mase":np.nan,
                "micro_mae":np.nan,"micro_smape":np.nan,"micro_mase":np.nan}
    # Macro = unweighted mean across tickers
    macro = per_ticker_df[["mae","smape","mase"]].mean().to_dict()
    # Micro/weighted by n (pooled)
    w = per_ticker_df["n"].to_numpy()
    micro = {
        "micro_mae": float(np.average(per_ticker_df["mae"], weights=w)),
        "micro_smape": float(np.average(per_ticker_df["smape"], weights=w)),
        "micro_mase": float(np.average(per_ticker_df["mase"], weights=w)),
    }
    return {f"macro_{k}": float(v) for k,v in macro.items()} | micro

# Run on 2 splits in class; you can expand later
import pathlib, json
pathlib.Path("reports").mkdir(exist_ok=True)
rows=[]
for sid, split in enumerate(splits[:2], start=1):
    a,b,c,d = split
    tr_idx, va_idx = splits_to_indices(returns, split)
    tr = preds_all.loc[tr_idx].copy()
    va = preds_all.loc[va_idx].copy()
    # Per-ticker metrics for two baselines
    pt_naive = per_ticker_metrics(va, tr, method="naive")
    pt_s     = per_ticker_metrics(va, tr, method="s")
    agg_naive = aggregate_across_tickers(pt_naive)
    agg_s     = aggregate_across_tickers(pt_s)
    # Save per-split, per-ticker
```

```
    pt_naive.to_csv(f"reports/baseline_naive_split{sid}.csv", index=False)
    pt_s.to_csv(f"reports/baseline_s{seasonality}_split{sid}.csv", index=False)
    rows.append({
        "split": sid,
        "train_range": f"{a.date()}→{b.date()}",
        "val_range": f"{c.date()}→{d.date()}",
        "method": "naive", **agg_naive
    })
    rows.append({
        "split": sid,
        "train_range": f"{a.date()}→{b.date()}",
        "val_range": f"{c.date()}→{d.date()}",
        "method": f"s{seasonality}", **agg_s
    })

summary = pd.DataFrame(rows)
summary.to_csv("reports/baselines_rollingorigin_summary.csv", index=False)
summary
```

### 17.4.5 4) Quick sanity assertions (no overlap; embargo honored)

```
def check_no_overlap(df, split):
    a,b,c,d = split
    assert b < c, f"Embargo violation: train_end {b} >= val_start {c}"
    tr_idx, va_idx = splits_to_indices(df, split)
    assert set(tr_idx).isdisjoint(set(va_idx))
    return True

all(check_no_overlap(returns, s) for s in splits[:2]), len(summary)
```

---

## 17.5 Wrap-up (10 min)

- You now have a **date-based rolling-origin splitter** with an **embargo**, and **baseline metrics** that set a credible reference.
- **MASE** uses a **training-window naive** as scale (per ticker), so you can read "<1 is better than naive" at a glance.

- Aggregation: report both **macro** (per-ticker average) and **micro/weighted** (pooled).

---

## 17.6 Homework (due before Session 16)

**Goal:** Build a small CLI to reproduce these baselines over **all splits**, then generate per-ticker & aggregated tables.

### 17.6.1 Part A — Script: `scripts/baselines_eval.py`

```
#!/usr/bin/env python
from __future__ import annotations
import argparse, numpy as np, pandas as pd
from pathlib import Path

def mae(y,yhat): return float(np.mean(np.abs(np.asarray(y)-np.asarray(yhat))))
def smape(y,yhat,eps=1e-8):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(2*np.abs(y-yhat)/(np.abs(y)+np.abs(yhat)+eps)))
def mase(y_true, y_pred, y_train_true, y_train_naive):
    return float(mae(y_true, y_pred) / (mae(y_train_true, y_train_naive)+1e-12))

def make_splits(dates, train_min, val_size, step, embargo):
    u = np.array(sorted(pd.to_datetime(dates.unique()))); n=len(u); out=[]; i=train_min-1
    while True:
        if i>=n: break
        a,b = u[0], u[i]; vs = i + embargo + 1; ve = vs + val_size - 1
        if ve>=n: break
        out.append((a,b,u[vs],u[ve])); i += step
    return out

def add_preds(df, s):
    out = df.copy()
    out["yhat_naive"] = out.groupby("ticker")["log_return"].transform(lambda x: x)
    out["yhat_s"] = out.groupby("ticker")["log_return"].transform(lambda x: x.shift(s-1)) if
    return out

def per_ticker(df_val, df_train, method, s):
```

```
    col = "yhat_naive" if method=="naive" else "yhat_s"
    rows=[]
    for tkr, g in df_val.groupby("ticker"):
        gv = g.dropna(subset=["r_1d", col])
        if len(gv)==0: continue
        gt = df_train[df_train["ticker"]==tkr].dropna(subset=["r_1d"])
        gt_pred = gt["log_return"] if method=="naive" else gt["log_return"].shift(s-1)
        gt_pred = gt_pred.loc[gt.index]
        y_tr = gt["r_1d"].to_numpy(); yhat_tr = gt_pred.to_numpy()
        y = gv["r_1d"].to_numpy(); yhat = gv[col].to_numpy()
        rows.append({"ticker":tkr,"n":int(len(y)),
                     "mae": mae(y,yhat),
                     "smape": smape(y,yhat),
                     "mase": mase(y,yhat,y_tr,yhat_tr)})
    return pd.DataFrame(rows)

def agg(pt):
    if pt.empty: return {"macro_mae":np.nan,"macro_smape":np.nan,"macro_mase":np.nan,
                         "micro_mae":np.nan,"micro_smape":np.nan,"micro_mase":np.nan}
    macro = pt[["mae","smape","mase"]].mean().to_dict()
    w = pt["n"].to_numpy()
    micro = {
        "micro_mae": float(np.average(pt["mae"], weights=w)),
        "micro_smape": float(np.average(pt["smape"], weights=w)),
        "micro_mase": float(np.average(pt["mase"], weights=w)),
    }
    return {f"macro_{k}": float(v) for k,v in macro.items()} | micro

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--returns", default="data/processed/returns.parquet")
    ap.add_argument("--seasonality", type=int, default=5)
    ap.add_argument("--train-min", type=int, default=252)
    ap.add_argument("--val-size", type=int, default=63)
    ap.add_argument("--step", type=int, default=63)
    ap.add_argument("--embargo", type=int, default=5)
    ap.add_argument("--out-summary", default="reports/baselines_rollingorigin_summary.csv")
    ap.add_argument("--out-per-ticker", default="reports/baselines_per_ticker_split{sid}_{me
    args = ap.parse_args()

    df = pd.read_parquet(args.returns).sort_values(["ticker","date"]).reset_index(drop=True)
    splits = make_splits(df["date"], args.train_min, args.val_size, args.step, args.embargo)
```

```python
    pred = add_preds(df, args.seasonality)

    rows=[]
    for sid, (a,b,c,d) in enumerate(splits, start=1):
        tr = pred[(pred["date"]>=a)&(pred["date"]<=b)]
        va = pred[(pred["date"]>=c)&(pred["date"]<=d)]
        for method in ["naive","s"]:
            pt = per_ticker(va, tr, method, args.seasonality)
            Path("reports").mkdir(exist_ok=True)
            pt.to_csv(args.out_per_ticker.format(sid=sid, method=method), index=False)
            rows.append({"split":sid,"train_range":f"{a.date()}→{b.date()}","val_range":f"{c
                         "method":"naive" if method=="naive" else f"s{args.seasonality}", **
    pd.DataFrame(rows).to_csv(args.out_summary, index=False)
    print("Wrote", args.out_summary, "and per-ticker CSVs.")

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

Make executable & run:

```bash
%%bash
chmod +x scripts/baselines_eval.py
python scripts/baselines_eval.py --seasonality 5
```

### 17.6.2 Part B — Plot a tiny, informative results figure

```python
import pandas as pd, matplotlib.pyplot as plt, pathlib
pathlib.Path("docs/figs").mkdir(parents=True, exist_ok=True)

summary = pd.read_csv("reports/baselines_rollingorigin_summary.csv")
plt.figure(figsize=(6,3.5))
for method, g in summary.groupby("method"):
    plt.plot(g["split"], g["micro_mae"], marker="o", label=f"{method} micro MAE")
plt.xlabel("Split"); plt.ylabel("MAE"); plt.title("Baseline MAE across splits")
plt.legend(); plt.tight_layout()
plt.savefig("docs/figs/baselines_mae_splits.png", dpi=200)
"Saved docs/figs/baselines_mae_splits.png"
```

### 17.6.3 Part C — Add a quick test to protect the splitter

```python
# tests/test_rolling_splitter.py
import pandas as pd, numpy as np
from datetime import timedelta

def make_splits(dates, train_min, val_size, step, embargo):
    u = np.array(sorted(pd.to_datetime(dates.unique()))); n=len(u); out=[]; i=train_min-1
    while True:
        if i>=n: break
        a,b = u[0], u[i]; vs=i+embargo+1; ve=vs+val_size-1
        if ve>=n: break
        out.append((a,b,u[vs],u[ve])); i+=step
    return out

def test_embargo_and_order():
    dates = pd.bdate_range("2024-01-01", periods=400)
    s = make_splits(pd.Series(dates), 252, 63, 63, 5)
    assert all(b < c for (a,b,c,d) in s), "Embargo/order violated"
    # Splits should move forward
    assert len(s) >= 2 and s[1][1] > s[0][1]
```

Run:

```bash
%%bash
pytest -q -k rolling_splitter
```

### 17.6.4 Part D — (Optional) Makefile targets

```
.PHONY: baselines
baselines: ## Evaluate naive & seasonal-naive baselines across all splits
\tpython scripts/baselines_eval.py --seasonality 5
```

---

## 17.7 Instructor checklist (before class)

- Ensure `returns.parquet` exists or fallback works.

- Be ready to whiteboard **why** the seasonal naïve for daily data uses `s=5`.
- Emphasize **MASE scale from TRAIN** and **macro vs micro** aggregation.

## 17.8 Emphasize while teaching

- **Define the problem first** (H, step, splits); metrics only make sense after framing.
- **MASE $< 1$** better than naïve; report both macro & micro.
- **Embargo** helps mitigate adjacency leakage; keep it small but nonzero.

## 17.9 Grading (pass/revise)

- Rolling-origin splitter implemented and used (train/val ranges printed).
- Reports written: `baselines_rollingorigin_summary.csv` and per-ticker CSVs per split & method.
- Metrics include **MAE**, **sMAPE**, **MASE**; aggregation includes **macro** and **micro**.
- A test asserts basic splitter properties (no overlap; forward progress).

You now have clear **framing and metrics** for your project. In Session 16, you'll fit **classical baselines** (e.g., lags-only linear, ARIMA/ETS quick sketches) and log them in the same results table schema.

# 18 Session 16

Below is a complete lecture package for **Session 16 — Classical Baselines** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. In class you'll train a **lags-only linear regressor per ticker** and compare it to the **naive** and **seasonal-naive** baselines from Session 15. You'll also see a short, optional **ARIMA** demo and log results in a consistent schema for future comparison.

> **Educational use only — not trading advice.** Assumes your repo (e.g., `unified-stocks-teamX`) with `data/processed/returns.parquet` and `data/processed/features_v1.parquet` from Sessions 9–10. Cells include safe fallbacks if some files are missing.

---

## 18.1 Session 16 — Classical baselines (75 min)

### 18.1.1 Learning goals

By the end of class, students can:

1. Fit a **per-ticker** lags-only linear regressor to predict **next-day log return** $r_{t+1}$.
2. Evaluate models with **MAE, sMAPE, MASE** using the **rolling-origin splits** (with embargo) from Session 15.
3. Log results in a **consistent table schema** for per-ticker and split-level summaries.
4. Understand **ARIMA** at a glance and its common pitfalls (optional demo).

---

## 18.2 Agenda (75 min)

- **(10 min)** Slides: where classical models fit; pitfalls with ARIMA; cross-sectional regressors
- **(10 min)** Slides: results table schema & comparison to baselines
- **(35 min) In-class lab**: train per-ticker **Linear (lags-only)** → evaluate across 2 splits → compare to naive/seasonal-naive → log CSVs
- **(10 min)** Wrap-up + homework brief
- **(10 min)** Buffer

---

## 18.3 Slides / talking points

### 18.3.1 Why "classical" now?

- Creates a **credible, strong baseline** against naive that's still transparent.
- Supports **fast iteration** and helps you debug feature definitions before deep models.

### 18.3.2 Lags-only linear regressor

- **Features** at time $t$: `lag1`, `lag2`, `lag3` (i.e., past returns), optionally a few stable stats (`roll_std_20`, `zscore_20`).
- **Target**: `r_1d` (next-day log return).
- Fit **per ticker** to avoid cross-sectional leakage for now.

### 18.3.3 ARIMA 60-second pitfall tour

- Stationarity: **fit on returns**, not prices (unless differencing).
- Evaluation: **re-fit only on train**; generate **one-step-ahead** forecasts on val, updating state **without peeking**.
- Over-differencing & mis-specified seasonal terms → bad bias.
- Computational cost grows with grid search; keep demo tiny.

### 18.3.4 Results table schema (consistent across sessions)

- **Per-split summary**: `split, train_range, val_range, model, macro_mae, macro_smape, macro_mase, micro_mae, micro_smape, micro_mase`
- **Per-ticker metrics**: `split, ticker, n, model, mae, smape, mase`

---

## 18.4 In-class lab (35 min, Colab-friendly)

Run each block as its **own cell**. Update `REPO_NAME` if needed.

### 18.4.1 0) Setup & data (with fallbacks)

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"  # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, numpy as np, pandas as pd
from pathlib import Path
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in ["data/raw","data/processed","reports","models","scripts","tests"]:
    Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())

# Load returns; if missing, synthesize
rpath = Path("data/processed/returns.parquet")
if rpath.exists():
    returns = pd.read_parquet(rpath)
else:
    rng = np.random.default_rng(0)
    dates = pd.bdate_range("2022-01-03", periods=360)
    rows=[]
    for t in ["AAPL","MSFT","GOOGL","AMZN","NVDA"]:
        eps = rng.normal(0,0.012, size=len(dates)).astype("float32")
```

```
        adj = 100*np.exp(np.cumsum(eps))
        df = pd.DataFrame({
            "date": dates, "ticker": t,
            "adj_close": adj.astype("float32"),
            "log_return": np.r_[np.nan, np.diff(np.log(adj))].astype("float32")
        })
        df["r_1d"] = df["log_return"].shift(-1)
        df["weekday"] = df["date"].dt.weekday.astype("int8")
        df["month"]   = df["date"].dt.month.astype("int8")
        rows.append(df)
    returns = pd.concat(rows, ignore_index=True).dropna().reset_index(drop=True)
    returns["ticker"] = returns["ticker"].astype("category")
    returns.to_parquet(rpath, index=False)

# Load features_v1 or derive minimal lags from returns if missing
fpath = Path("data/processed/features_v1.parquet")
if fpath.exists():
    feats = pd.read_parquet(fpath)
else:
    # Minimal lags derived just from returns
    feats = returns.sort_values(["ticker","date"]).copy()
    for k in [1,2,3]:
        feats[f"lag{k}"] = feats.groupby("ticker")["log_return"].shift(k)
    feats = feats.dropna(subset=["lag1","lag2","lag3","r_1d"]).reset_index(drop=True)

# Harmonize
feats["date"] = pd.to_datetime(feats["date"])
feats["ticker"] = feats["ticker"].astype("category")
feats = feats.sort_values(["ticker","date"]).reset_index(drop=True)
feats.head()
```

### 18.4.2 1) Rolling-origin date splits (reuse Session 15 logic)

```
def make_rolling_origin_splits(dates, train_min=252, val_size=63, step=63, embargo=5):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    splits=[]; i = train_min-1; n=len(u)
    while True:
        if i>=n: break
        a,b = u[0], u[i]; vs=i+embargo+1; ve=vs+val_size-1
        if ve>=n: break
```

```
        splits.append((a,b,u[vs],u[ve])); i+=step
    return splits

splits = make_rolling_origin_splits(feats["date"], 252, 63, 63, 5)
len(splits), splits[:2]
```

### 18.4.3 2) Metrics & baselines (from Session 15)

```
def mae(y, yhat):
    y = np.asarray(y); yhat = np.asarray(yhat);
    return float(np.mean(np.abs(y - yhat)))

def smape(y, yhat, eps=1e-8):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(2.0*np.abs(y - yhat)/(np.abs(y)+np.abs(yhat)+eps)))

def mase(y_true, y_pred, y_train_true, y_train_naive):
    scale = mae(y_train_true, y_train_naive) + 1e-12
    return float(mae(y_true, y_pred)/scale)

def add_baseline_preds(df: pd.DataFrame, seasonality:int=5) -> pd.DataFrame:
    out = df.copy()
    out["yhat_naive"] = out.groupby("ticker")["log_return"].transform(lambda s: s)
    out["yhat_s"] = out.groupby("ticker")["log_return"].transform(lambda s: s.shift(seasonali
    return out
```

### 18.4.4 3) Per-ticker lags-only LinearRegression (fit only on each split's TRAIN)

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

# Choose features (lags only for in-class lab)
XCOLS = [c for c in ["lag1","lag2","lag3"] if c in feats.columns]
assert XCOLS, "No lag features found. Ensure features_v1 or fallback creation ran."

def fit_predict_lin_lags(train_df, val_df):
    """Fit per-ticker pipeline(StandardScaler, LinearRegression) on TRAIN; predict on VAL."""
```

```
    preds=[]
    for tkr, tr in train_df.groupby("ticker"):
        va = val_df[val_df["ticker"]==tkr]
        if len(tr)==0 or len(va)==0:
            continue
        pipe = Pipeline([("scaler", StandardScaler(with_mean=True, with_std=True)),
                         ("lr", LinearRegression())])
        pipe.fit(tr[XCOLS].values, tr["r_1d"].values)
        yhat = pipe.predict(va[XCOLS].values)
        out = va[["date","ticker","r_1d","log_return"]].copy()
        out["yhat_linlags"] = yhat.astype("float32")
        preds.append(out)
    return pd.concat(preds, ignore_index=True) if preds else pd.DataFrame(columns=["date","ti
```

### 18.4.5 4) Evaluate across the first 2 splits; compare to naive/seasonal-naive

```
seasonality = 5
feats_baseline = add_baseline_preds(feats, seasonality=seasonality)

def per_ticker_metrics(df_val_pred, df_train, method_col):
    rows=[]
    for tkr, gv in df_val_pred.groupby("ticker"):
        if method_col not in gv:
            continue
        gv = gv.dropna(subset=["r_1d", method_col])
        if len(gv)==0:
            continue
        # TRAIN scale for MASE
        gt = df_train[df_train["ticker"]==tkr].dropna(subset=["r_1d"])
        gt_naive = gt["log_return"] if "yhat_s" not in method_col else gt["log_return"].shif
        gt_naive = gt_naive.loc[gt.index]
        rows.append({
            "ticker": tkr,
            "n": int(len(gv)),
            "mae": mae(gv["r_1d"], gv[method_col]),
            "smape": smape(gv["r_1d"], gv[method_col]),
            "mase": mase(gv["r_1d"], gv[method_col], gt["r_1d"], gt_naive),
        })
    return pd.DataFrame(rows)
```

```
def summarize_split(feats_frame, sid, split, save_prefix="linlags"):
    a,b,c,d = split
    tr = feats_frame[(feats_frame["date"]>=a)&(feats_frame["date"]<=b)].copy()
    va = feats_frame[(feats_frame["date"]>=c)&(feats_frame["date"]<=d)].copy()
    # Predictions
    val_pred = fit_predict_lin_lags(tr, va)
    # Attach baseline preds on val slice
    va_base = add_baseline_preds(va, seasonality=seasonality)
    val_pred = val_pred.merge(va_base[["date","ticker","yhat_naive","yhat_s"]], on=["date","

    # Per-ticker metrics
    pt_lin = per_ticker_metrics(val_pred, tr, "yhat_linlags"); pt_lin["model"] = "lin_lags"
    pt_nav = per_ticker_metrics(val_pred.rename(columns={"yhat_naive":"yhat_linlags"}), tr, 
    pt_sea = per_ticker_metrics(val_pred.rename(columns={"yhat_s":"yhat_linlags"}), tr, "yha

    # Save per-ticker
    out_pt = pd.concat([pt_lin.assign(split=sid), pt_nav.assign(split=sid), pt_sea.assign(sp
    out_pt.to_csv(f"reports/{save_prefix}_per_ticker_split{sid}.csv", index=False)

    # Aggregate
    def agg(df):
        if df.empty:
            return {"macro_mae":np.nan,"macro_smape":np.nan,"macro_mase":np.nan,"micro_mae":
        macro = df[["mae","smape","mase"]].mean().to_dict()
        w = df["n"].to_numpy()
        micro = {"micro_mae": float(np.average(df["mae"], weights=w)),
                 "micro_smape": float(np.average(df["smape"], weights=w)),
                 "micro_mase": float(np.average(df["mase"], weights=w))}
        return {f"macro_{k}":float(v) for k,v in macro.items()} | micro

    rows=[]
    for name, pt in [("lin_lags", pt_lin), ("naive", pt_nav), (f"s{seasonality}", pt_sea)]:
        rows.append({"split":sid, "train_range": f"{a.date()}→{b.date()}",
                     "val_range": f"{c.date()}→{d.date()}",
                     "model":name, **agg(pt)})
    return pd.DataFrame(rows)

# Run on first 2 splits in class
summary_frames=[]
for sid, split in enumerate(splits[:2], start=1):
    sf = summarize_split(feats_baseline, sid, split, save_prefix="linlags")
    summary_frames.append(sf)
```

```
summary = pd.concat(summary_frames, ignore_index=True)
summary.to_csv("reports/linlags_summary_splits12.csv", index=False)
summary
```

### 18.4.6 5) (Optional) Tiny ARIMA demo on one ticker for the first split

```
# Optional: quick ARIMA(1,0,0) demo predicting r_{t+1} on val for a single ticker
try:
    from statsmodels.tsa.arima.model import ARIMA
    import warnings; warnings.filterwarnings("ignore")
    a,b,c,d = splits[0]
    tkr = feats["ticker"].cat.categories[0]
    tr = feats[(feats["ticker"]==tkr) & (feats["date"]>=a) & (feats["date"]<=b)]
    va = feats[(feats["ticker"]==tkr) & (feats["date"]>=c) & (feats["date"]<=d)]
    # Fit on TRAIN returns only (endog = log_return). Predict one-step ahead for VAL dates.
    model = ARIMA(tr["log_return"].to_numpy(), order=(1,0,0))
    res = model.fit()
    # Forecast length = len(va), one-step-ahead with dynamic=False updates internally
    # (For strict no-peek rolling one-step, loop and append val true values; here we keep der
    fc = res.forecast(steps=len(va))
    arima_mae = mae(va["r_1d"], fc)  # compare against next-day return
    float(arima_mae)
except Exception as e:
    print("ARIMA demo skipped:", e)
```

ARIMA is **optional** and **slow** on large loops. If you try it per ticker/per split, keep the dataset tiny.

---

## 18.5 Wrap-up (10 min)

- You trained a **per-ticker lags-only linear** model and compared it fairly to **naive** and **seasonal-naive** using the **same splits** and **MASE scale** (from the train window).
- You logged results in a **stable schema** that you'll reuse for future models (LSTM / Transformer).
- ARIMA can be illustrative but is often **fragile + slower**; treat it as optional for your project scale.

---

## 18.6 Homework (due before next session)

**Goal:** 1) Run the linear lags baseline across **all splits**; 2) Write your **first model card** (Quarto) for the classical baseline.

### 18.6.1 Part A — CLI script to evaluate Linear-Lags across *all* splits

```
# scripts/eval_linlags.py
#!/usr/bin/env python
from __future__ import annotations
import argparse, numpy as np, pandas as pd
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from pathlib import Path

def mae(y,yhat): return float(np.mean(np.abs(np.asarray(y)-np.asarray(yhat))))
def smape(y,yhat,eps=1e-8):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(2*np.abs(y-yhat)/(np.abs(y)+np.abs(yhat)+eps)))
def mase(y_true, y_pred, y_train_true, y_train_naive):
    return float(mae(y_true, y_pred) / (mae(y_train_true, y_train_naive)+1e-12))

def make_splits(dates, train_min, val_size, step, embargo):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    splits=[]; i=train_min-1; n=len(u)
    while True:
        if i>=n: break
        a,b = u[0], u[i]; vs=i+embargo+1; ve=vs+val_size-1
        if ve>=n: break
        splits.append((a,b,u[vs],u[ve])); i+=step
    return splits

def add_baselines(df, seasonality):
    out = df.copy()
    out["yhat_naive"] = out.groupby("ticker")["log_return"].transform(lambda s: s)
    out["yhat_s"] = out.groupby("ticker")["log_return"].transform(lambda s: s.shift(seasonali
```

```
    return out

def fit_predict_lin(train_df, val_df, xcols):
    from sklearn.linear_model import LinearRegression
    from sklearn.preprocessing import StandardScaler
    from sklearn.pipeline import Pipeline
    preds=[]
    for tkr, tr in train_df.groupby("ticker"):
        va = val_df[val_df["ticker"]==tkr]
        if len(tr)==0 or len(va)==0: continue
        pipe = Pipeline([("scaler", StandardScaler()), ("lr", LinearRegression())])
        pipe.fit(tr[xcols].values, tr["r_1d"].values)
        yhat = pipe.predict(va[xcols].values)
        out = va[["date","ticker","r_1d","log_return"]].copy()
        out["yhat_linlags"] = yhat
        preds.append(out)
    return pd.concat(preds, ignore_index=True) if preds else pd.DataFrame()

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--features", default="data/processed/features_v1.parquet")
    ap.add_argument("--seasonality", type=int, default=5)
    ap.add_argument("--train-min", type=int, default=252)
    ap.add_argument("--val-size", type=int, default=63)
    ap.add_argument("--step", type=int, default=63)
    ap.add_argument("--embargo", type=int, default=5)
    ap.add_argument("--xcols", nargs="+", default=["lag1","lag2","lag3"])
    ap.add_argument("--out-summary", default="reports/linlags_summary.csv")
    ap.add_argument("--out-per-ticker", default="reports/linlags_per_ticker_split{sid}.csv")
    args = ap.parse_args()

    df = pd.read_parquet(args.features).sort_values(["ticker","date"]).reset_index(drop=True
    df["ticker"] = df["ticker"].astype("category")
    splits = make_splits(df["date"], args.train_min, args.val_size, args.step, args.embargo)
    df = add_baselines(df, args.seasonality)

    rows=[]
    for sid, (a,b,c,d) in enumerate(splits, start=1):
        tr = df[(df["date"]>=a)&(df["date"]<=b)]
        va = df[(df["date"]>=c)&(df["date"]<=d)]
        val_pred = fit_predict_lin(tr, va, args.xcols)
        va = va.merge(val_pred[["date","ticker","yhat_linlags"]], on=["date","ticker"], how='
```

```
        # per-ticker
        pts=[]
        for tkr, gv in va.groupby("ticker"):
            gv = gv.dropna(subset=["r_1d","yhat_linlags"])
            if len(gv)==0: continue
            gt = tr[tr["ticker"]==tkr].dropna(subset=["r_1d"])
            gt_naive = gt["log_return"]  # scale comparator for MASE
            pts.append({"ticker":tkr,"n":int(len(gv)),
                        "mae": mae(gv["r_1d"], gv["yhat_linlags"]),
                        "smape": smape(gv["r_1d"], gv["yhat_linlags"]),
                        "mase": mase(gv["r_1d"], gv["yhat_linlags"], gt["r_1d"], gt_naive)})
        pt = pd.DataFrame(pts)
        Path("reports").mkdir(exist_ok=True)
        pt.assign(split=sid, model="lin_lags").to_csv(args.out_per_ticker.format(sid=sid), in

        # aggregate
        if not pt.empty:
            macro = pt[["mae","smape","mase"]].mean().to_dict()
            w = pt["n"].to_numpy()
            micro = {"micro_mae": float(np.average(pt["mae"], weights=w)),
                     "micro_smape": float(np.average(pt["smape"], weights=w)),
                     "micro_mase": float(np.average(pt["mase"], weights=w))}
        else:
            macro = {"mae":np.nan,"smape":np.nan,"mase":np.nan}
            micro = {"micro_mae":np.nan,"micro_smape":np.nan,"micro_mase":np.nan}
        rows.append({"split":sid,"train_range":f"{a.date()}→{b.date()}","val_range":f"{c.date
                     "model":"lin_lags", "macro_mae":float(macro["mae"]), "macro_smape":float
                     **micro})

    pd.DataFrame(rows).to_csv(args.out_summary, index=False)
    print("Wrote", args.out_summary)

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

Make executable & run:

```
%%bash
chmod +x scripts/eval_linlags.py
python scripts/eval_linlags.py --xcols lag1 lag2 lag3
```

### 18.6.2 Part B — Quarto Model Card for the Linear-Lags baseline

Create `docs/model_card_linear.qmd`:

~~~
---
title: "Model Card - Linear Lags (Per-Ticker)"
format:
  html:
    theme: cosmo
    toc: true
params:
  model_name: "Linear Lags (per-ticker)"
  data: "features_v1.parquet"
---

> **Educational use only - not trading advice.** Predicts next-day log return \(r_{t+1}\) usi

## Overview

- **Model:** Per-ticker linear regression with features: `lag1`, `lag2`, `lag3`.
- **Data:** `features_v1.parquet` (Session 10).
- **Splits:** Expanding, quarterly val, 5-day embargo (Session 15).
- **Baselines:** Naive and seasonal-naive \(s=5\).

## Metrics (across splits)

::: {.cell execution_count=1}
````` {.python .cell-code}
import pandas as pd
df = pd.read_csv("reports/linlags_summary.csv")
df
~~~

## 18.7 Discussion

- **Assumptions:** Linear relation to recent returns; stationarity at return level.
- **Strengths:** Fast, interpretable, leakage-resistant with proper splits.
- **Failure modes:** Regime shifts; volatility spikes; nonlinearity.
- **Ethics:** Educational; not suitable for trading.

`Render (if Quarto is available):`

````
```bash
quarto render docs/model_card_linear.qmd
````

### 18.7.1 Part C — Quick test to safeguard results shape

```python
# tests/test_linlags_results.py
import pandas as pd, os

def test_linlags_summary_exists_and_columns():
    assert os.path.exists("reports/linlags_summary.csv")
    df = pd.read_csv("reports/linlags_summary.csv")
    need = {"split","model","macro_mae","micro_mae"}
    assert need.issubset(df.columns)
```

:::

Run:

```bash
%%bash
pytest -q -k linlags_results
```

### 18.7.2 Part D — (Optional) Extend features or add Ridge

- Try `--xcols lag1 lag2 lag3 roll_std_20 zscore_20` (if present in `features_v1`).
- Swap `LinearRegression` for `Ridge(alpha=1.0)`; log and compare.

---

## 18.8 Instructor checklist (before class)

- Verify `features_v1.parquet` has `lag1..lag3` or the fallback cell creates them.
- Dry-run the 2-split demo; ensure total runtime $<$ 5–6 minutes.
- Optionally prepare an ARIMA demo on **one** ticker to illustrate pitfalls.
```

## 18.9 Emphasize while teaching

- Keep **splits identical** across models for fair comparison.
- **MASE $< 1$** your model beats naive on train-scale; report macro & micro.
- Linear lags are a **transparent baseline**—use them to validate your entire pipeline.

## 18.10 Grading (pass/revise)

- `scripts/eval_linlags.py` runs and writes `reports/linlags_summary.csv` + per-ticker CSVs.
- Model card exists and renders (locally or in CI artifact).
- Tests for results table shape pass.
- Results show a reasonable comparison against naive/seasonal-naive.

You now have a **solid classical baseline** with a reproducible evaluation and reporting workflow—perfect for benchmarking upcoming neural models.

# 19 Session 17 — Feature Timing, Biases & Leakage

Below is a complete lecture package for **Session 17 — Feature Timing, Biases & Leakage** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class lab with copy-paste code**, and **homework with copy-paste code**. In class you'll **freeze a static ticker universe** (avoid survivorship bias), **formalize label definitions** (t+1 and multi-step), and add a **leakage test suite** that fails if any feature at time *t* uses information from *t+1* or later.

> **Educational use only — not trading advice.** Assumes your Drive-mounted repo (e.g., `unified-stocks-teamX`) with `data/processed/returns.parquet` and `data/processed/features_v1.parquet` from Sessions 9–10. Cells include safe fallbacks when files are missing.

---

## 19.1 Session 17 — Feature Timing, Biases & Leakage (75 min)

### 19.1.1 Learning goals

By the end of class, students can:

1. Explain and **avoid look-ahead** and **survivorship** biases.
2. Freeze and use a **static ticker universe** chosen from the **train window** (not the whole history).
3. Define labels correctly (e.g., **t+1** and **t+5**) and verify them with tests.
4. Add **leakage tests** that recompute trusted features and fail on any future-peek.

---

## 19.2 Agenda (75 min)

- **(10 min)** Slides: what leakage looks like; examples; how it sneaks in
- **(10 min)** Slides: survivorship bias (today's constituents  past reality); freezing a universe
- **(10 min)** Slides: label definitions (t+1, multi-step) and alignment rules
- **(35 min) In-class lab**:
    1. Freeze a static universe from the first split's train window
    2. Add leakage tests that recompute known-good features
    3. Add multi-step labels (e.g., t+5) with tests
- **(10 min)** Wrap-up & homework brief

---

## 19.3 Slides / talking points (drop into your deck)

### 19.3.1 What is data leakage?

- **Look-ahead leakage:** using any info from *t+1* or later to compute features at *t* or to scale/normalize train and validation together.
- **Common culprits:** `shift(-1)` in features, global scaling fit on full data, forward-fill across split boundaries, using today's close to predict today's close.

### 19.3.2 Survivorship bias

- Using **today's index membership** to pick tickers for the past  drops delisted/removed names  **optimistically biased** results.
- **Cure:** freeze a **static universe** from the **training window** (e.g., all tickers with  252 observations by the end of the first train window). Save it and **filter by it** for all future experiments.

### 19.3.3 Label definitions (be explicit)

- **t+1 log return**: `r_1d = log_return.shift(-1)` per ticker (your Session-9 label).
- **t+5 log return** (multi-step): `r_5d = log_return.shift(-1) + … + log_return.shift(-5)` per ticker.
- Rules: labels come from **future**; features come from **t**. Splits with **embargo** reduce adjacency leakage.

---

## 19.4 In-class lab (35 min, Colab-friendly)

Run each block as its own cell. Update `REPO_NAME` as needed.

### 19.4.1 0) Setup & load data (with fallbacks)

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"   # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, numpy as np, pandas as pd
from pathlib import Path
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in ["data/raw","data/processed","data/static","reports","scripts","tests"]:
    Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())

# Load returns or synthesize a small fallback
rpath = Path("data/processed/returns.parquet")
if rpath.exists():
    returns = pd.read_parquet(rpath)
else:
    rng = np.random.default_rng(0)
    dates = pd.bdate_range("2022-01-03", periods=360)
    rows=[]
```

```
    for t in ["AAPL","MSFT","GOOGL","AMZN","NVDA","TSLA","META","NFLX"]:
        eps = rng.normal(0,0.012,size=len(dates)).astype("float32")
        adj = 100*np.exp(np.cumsum(eps))
        df = pd.DataFrame({
            "date": dates, "ticker": t,
            "adj_close": adj.astype("float32"),
            "log_return": np.r_[np.nan, np.diff(np.log(adj))].astype("float32")
        })
        df["r_1d"] = df["log_return"].shift(-1)
        df["weekday"] = df["date"].dt.weekday.astype("int8")
        df["month"]   = df["date"].dt.month.astype("int8")
        rows.append(df)
    returns = pd.concat(rows, ignore_index=True).dropna().reset_index(drop=True)
    returns["ticker"] = returns["ticker"].astype("category")
    returns.to_parquet(rpath, index=False)

# Load features_v1 or construct minimal lags for tests
fpath = Path("data/processed/features_v1.parquet")
if fpath.exists():
    feats = pd.read_parquet(fpath).sort_values(["ticker","date"]).reset_index(drop=True)
else:
    feats = returns.sort_values(["ticker","date"]).copy()
    for k in [1,2,3]:
        feats[f"lag{k}"] = feats.groupby("ticker")["log_return"].shift(k)
    feats["roll_mean_20"] = feats.groupby("ticker")["log_return"].rolling(20, min_periods=20
    feats["roll_std_20"]  = feats.groupby("ticker")["log_return"].rolling(20, min_periods=20
    feats["zscore_20"]    = (feats["log_return"] - feats["roll_mean_20"]) / (feats["roll_std
    feats = feats.dropna().reset_index(drop=True)

# Harmonize types
returns["date"] = pd.to_datetime(returns["date"])
feats["date"]   = pd.to_datetime(feats["date"])
returns["ticker"] = returns["ticker"].astype("category")
feats["ticker"]   = feats["ticker"].astype("category")
returns = returns.sort_values(["ticker","date"]).reset_index(drop=True)
feats   = feats.sort_values(["ticker","date"]).reset_index(drop=True)
returns.head(3), feats.head(3)
```

### 19.4.2 1) Freeze a static universe from the first split's train window

```
import numpy as np, pandas as pd

def make_rolling_origin_splits(dates, train_min=252, val_size=63, step=63, embargo=5):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    i = train_min - 1; splits=[]
    while True:
        if i >= len(u): break
        a,b = u[0], u[i]
        vs = i + embargo + 1
        ve = vs + val_size - 1
        if ve >= len(u): break
        splits.append((a,b,u[vs],u[ve]))
        i += step
    return splits

splits = make_rolling_origin_splits(returns["date"], train_min=252, val_size=63, step=63, emb
assert len(splits) >= 1, "Not enough history for a first split."
a,b,c,d = splits[0]
print("First train window:", a.date(), "→", b.date())

# Eligible = tickers with at least train_min rows by train_end (b)
train_slice = returns[(returns["date"]>=a) & (returns["date"]<=b)]
counts = train_slice.groupby("ticker").size()
eligible = counts[counts >= 252].index.sort_values()
universe = pd.DataFrame({"ticker": eligible})
univ_name = f"data/static/universe_{b.date()}.csv"
universe.to_csv(univ_name, index=False)
print("Saved static universe:", univ_name, "| tickers:", len(universe))
universe.head()
```

From now on, **filter** your data to `universe` before modeling/evaluation.

### 19.4.3 2) Apply the static universe to your features

```
feats_static = feats[feats["ticker"].isin(set(universe["ticker"]))].copy()
feats_static.to_parquet("data/processed/features_v1_static.parquet", compression="zstd", inde
print("Wrote data/processed/features_v1_static.parquet", feats_static.shape)
```

### 19.4.4 3) Add leakage tests that recompute trusted features & compare

Create a high-value test file that **fails** if any feature depends on future rows.

```
# tests/test_leakage_features.py
from __future__ import annotations
import numpy as np, pandas as pd
import pytest

SAFE_ROLL = 20

@pytest.fixture(scope="session")
def df():
    import pandas as pd
    import pathlib
    p = pathlib.Path("data/processed/features_v1_static.parquet")
    if not p.exists():
        p = pathlib.Path("data/processed/features_v1.parquet")
    df = pd.read_parquet(p).sort_values(["ticker","date"]).reset_index(drop=True)
    df["date"] = pd.to_datetime(df["date"])
    return df

def test_label_definition_r1d(df):
    for tkr, g in df.groupby("ticker"):
        assert g["r_1d"].iloc[:-1].equals(g["log_return"].iloc[1:]), f"r_1d mismatch for {tk

def _recompute_safe(g: pd.DataFrame) -> pd.DataFrame:
    # Recompute causal features using only <= t information
    out = pd.DataFrame(index=g.index)
    s = g["log_return"]
    out["lag1"] = s.shift(1)
    out["lag2"] = s.shift(2)
    out["lag3"] = s.shift(3)
    rm = s.rolling(SAFE_ROLL, min_periods=SAFE_ROLL).mean()
    rs = s.rolling(SAFE_ROLL, min_periods=SAFE_ROLL).std()
    out["roll_mean_20"] = rm
    out["roll_std_20"]  = rs
    out["zscore_20"]    = (s - rm) / (rs + 1e-8)
    # EWM & expanding if present
    out["exp_mean"] = s.expanding(min_periods=SAFE_ROLL).mean()
    out["exp_std"]  = s.expanding(min_periods=SAFE_ROLL).std()
    out["ewm_mean_20"] = s.ewm(span=20, adjust=False).mean()
    out["ewm_std_20"]  = s.ewm(span=20, adjust=False).std()
```

```
    # RSI(14) if adj_close present
    if "adj_close" in g:
        delta = g["adj_close"].diff()
        up = delta.clip(lower=0).ewm(alpha=1/14, adjust=False).mean()
        dn = (-delta.clip(upper=0)).ewm(alpha=1/14, adjust=False).mean()
        rs = up / (dn + 1e-12)
        out["rsi_14"] = 100 - (100/(1+rs))
    return out

@pytest.mark.parametrize("col", ["lag1","lag2","lag3","roll_mean_20","roll_std_20","zscore_2
def test_features_match_causal_recompute(df, col):
    if col not in df.columns:
        pytest.skip(f"{col} not present")
    # Compare per ticker to avoid cross-group alignment issues
    for tkr, g in df.groupby("ticker", sort=False):
        ref = _recompute_safe(g)
        if col not in ref.columns:
            continue
        a = g[col].to_numpy()
        b = ref[col].to_numpy()
        # Allow NaNs at the start; compare where both finite
        mask = np.isfinite(a) & np.isfinite(b)
        if mask.sum() == 0:
            continue
        diff = np.nanmax(np.abs(a[mask] - b[mask]))
        assert float(diff) <= 1e-6, f"{col} deviates from causal recompute for {tkr}: max |Δ

def test_no_feature_equals_target(df):
    y = df["r_1d"].to_numpy()
    for col in df.select_dtypes(include=["float32","float64"]).columns:
        if col in {"r_1d","log_return"}:
            continue
        x = df[col].to_numpy()
        # Proportion of exact equality (within tiny tol) should not be high
        eq = np.isfinite(x) & np.isfinite(y) & (np.abs(x - y) < 1e-12)
        assert eq.mean() < 0.8, f"Suspicious: feature {col} equals target too often"
```

Run tests now:

```
!pytest -q tests/test_leakage_features.py
```

If a test fails, **fix the pipeline**, don't weaken the test.

### 19.4.5 4) Add multi-step labels (e.g., t+5) and tests

```
# scripts/make_multistep_labels.py
from __future__ import annotations
import pandas as pd, numpy as np
from pathlib import Path

def make_multistep(in_parquet="data/processed/returns.parquet", horizons=(5,)):
    df = pd.read_parquet(in_parquet).sort_values(["ticker","date"]).reset_index(drop=True)
    for H in horizons:
        # r_Hd = sum of next H log returns: shift(-1) ... shift(-H)
        s = df.groupby("ticker")["log_return"]
        acc = None
        for h in range(1, H+1):
            sh = s.shift(-h)
            acc = sh if acc is None else (acc + sh)
        df[f"r_{H}d"] = acc
    out = df
    Path("data/processed").mkdir(parents=True, exist_ok=True)
    out.to_parquet("data/processed/returns_multistep.parquet", compression="zstd", index=Fals
    print("Wrote data/processed/returns_multistep.parquet", out.shape)

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

Run it:

```
!python scripts/make_multistep_labels.py
```

Add a test for label correctness:

```
# tests/test_labels_multistep.py
import pandas as pd, numpy as np

def test_r5d_definition():
    df = pd.read_parquet("data/processed/returns_multistep.parquet").sort_values(["ticker","
    if "r_5d" not in df.columns:
        return
    for tkr, g in df.groupby("ticker"):
        lr = g["log_return"]
        r5 = sum(lr.shift(-h) for h in range(1,6))
```

```
        diff = (g["r_5d"] - r5).abs().max()
        assert float(diff) < 1e-10, f"r_5d misdefined for {tkr} (max |Δ|={diff})"
```

Run:

```
!pytest -q tests/test_labels_multistep.py
```

---

## 19.5 Wrap-up (10 min)

- **Static universe** removes **survivorship bias**: pick tickers with adequate history **by train end** and **stick to them**.
- Label definitions must be **explicit and tested** (t+1, t+5).
- Leakage tests **recompute causal features** and compare—if you accidentally used `shift(-1)` or cross-split fills, tests fail.

---

## 19.6 Homework (due before Session 18)

**Goal:** Document your evaluation protocol and ship a concise "leakage & bias" memo, plus a one-command audit.

### 19.6.1 Part A — Generate a protocol memo (`reports/eval_protocol.md`)

```
# scripts/write_eval_protocol.py
from __future__ import annotations
import pandas as pd, numpy as np
from pathlib import Path
from datetime import date

def make_rolling_origin_splits(dates, train_min=252, val_size=63, step=63, embargo=5):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    i = train_min - 1; out=[]
    while True:
```

```
        if i >= len(u): break
        a,b = u[0], u[i]; vs=i+embargo+1; ve=vs+val_size-1
        if ve >= len(u): break
        out.append((a,b,u[vs],u[ve])); i += step
    return out

def main():
    ret = pd.read_parquet("data/processed/returns.parquet").sort_values(["ticker","date"])
    splits = make_rolling_origin_splits(ret["date"])
    a,b,c,d = splits[0]
    # Universe info
    univ_files = sorted(Path("data/static").glob("universe_*.csv"))
    univ = univ_files[-1] if univ_files else None
    univ_count = pd.read_csv(univ).shape[0] if univ else ret["ticker"].nunique()
    md = []
    md += ["# Evaluation Protocol (Leakage-Aware)", ""]
    md += ["**Date:** " + date.today().isoformat(), ""]
    md += ["## Splits", f"- Train window (split 1): **{a.date()} → {b.date()}**",
           f"- Embargo: **5** business days", f"- Validation window: **{c.date()} → {d.date(
           f"- Step between origins: **63** business days", ""]
    md += ["## Static Universe", f"- Universe file: **{univ.name if univ else '(none)'}**",
           f"- Count: **{univ_count}** tickers",
           "- Selection rule: tickers with  252 obs by first train end; fixed for all splits.
    md += ["## Labels", "- `r_1d` = next-day log return `log_return.shift(-1)` per ticker.",
           "- `r_5d` (if used) = sum of `log_return.shift(-1..-5)`.", ""]
    md += ["## Leakage Controls",
           "- Features computed from   t only (rolling/ewm/expanding without negative shifts)
           "- No forward-fill across split boundaries; embargo = 5 days.",
           "- Scalers/normalizers fit on TRAIN only.",
           "- Tests: `tests/test_leakage_features.py`, `tests/test_labels_multistep.py`.", "
    md += ["## Caveats",
           "- Educational dataset; not investment advice.",
           "- Survivorship minimized via static universe; still subject to data vendor quirk
    Path("reports").mkdir(parents=True, exist_ok=True)
    Path("reports/eval_protocol.md").write_text("\n".join(md))
    print("Wrote reports/eval_protocol.md")

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

Run:

```
!python scripts/write_eval_protocol.py
```

### 19.6.2 Part B — One-command leakage audit target

Append to your `Makefile`:

```
.PHONY: leakage-audit
leakage-audit: ## Run leakage & label tests; write eval protocol
\tpytest -q tests/test_leakage_features.py tests/test_labels_multistep.py
\tpython scripts/write_eval_protocol.py
```

Then run:

```
make leakage-audit
```

### 19.6.3 Part C — Short memo (1–2 pages max)

- Open `reports/eval_protocol.md` and add **two paragraphs** in your own words:

  1. Why these splits and embargo are credible for your task.
  2. Where leakage could still hide (e.g., future macro revisions, implicit target leakage), and how you'd detect it.

  Submit the updated `reports/eval_protocol.md` and a screenshot of `make leakage-audit` passing.

### 19.6.4 Part D — (Optional) Quarto inclusion

Add this to your Quarto report:

`````
## Evaluation Protocol (Leakage-Aware)

::: {.cell execution_count=1}
````` {.python .cell-code}
from pathlib import Path
print(Path("reports/eval_protocol.md").read_text())
`````
`````

```
:::

---

## Instructor checklist (before class)
- Ensure `returns.parquet` and `features_v1.parquet` exist or fallback works.
- Intentionally create a leaked feature (e.g., `lag1 = log_return.shift(-1)`) on your copy to
- Decide an anchor date policy for universe freeze; today's lab uses **first split's train en

## Emphasize while teaching
- **Define labels first**, then prove features are **causal (  t)**.
- Freezing the **universe** is small effort with big impact on credibility.
- Tests are your **guardrails**-if they go red, **don't** relax them; fix the pipeline.

## Grading (pass/revise)
- `data/static/universe_YYYY-MM-DD.csv` created; `features_v1_static.parquet` filtered by it
- Leakage tests present and **green** on the clean pipeline; **red** if you inject a future-p
- `reports/eval_protocol.md` exists and includes student commentary.
- `make leakage-audit` runs without errors.

You now have a **credibility layer** on top of your data pipeline-ready to analyze regimes an
```

`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4ifQ== -->`{=html}

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiYm9va0
```

# Session 18 - Walk-forward + Regime Analysi

```
```````{.quarto-title-block template='/Users/yiwang/Applications/quarto/share/projects/book/p
---
title: Session 18 - Walk-forward + Regime Analysi

---
```

Below is a complete lecture package for **Session 18 — Walk-forward + Regime Analysis** (75 minutes). It includes a timed agenda, slide talking points, a **Colab-friendly in-class**
```

**lab with copy-paste code**, and **homework with copy-paste code**. You'll add **volatility regimes** to your rolling-origin evaluation (with embargo), compute **metrics by regime**, and produce **calibration plots** that reveal where baselines over/under-predict.

> **Educational use only — not trading advice.** Assumes your repo in Drive (e.g., `unified-stocks-teamX`) with `data/processed/returns.parquet` and `data/processed/features_v1.parquet`. If missing, the lab will synthesize a small fallback so you can run end-to-end.

---

## 19.7 Session 18 — Walk-forward + Regime Analysis (75 min)

### 19.7.1 Learning goals

By the end of class, students can:

1. Use **embargoed** rolling-origin splits (Session 15) and apply a **static universe** (Session 17) consistently.
2. Construct **volatility regimes** (low/med/high) from **rolling volatility** computed **causally** (  t), and set regime thresholds **using training-only** data per split.
3. Evaluate **MAE, sMAPE, MASE by regime**, with macro and micro aggregation.
4. Make **calibration plots** (binned predicted vs. realized returns) **by regime** and interpret them.

---

## 19.8 Agenda (75 min)

- **(10 min)** Slides: walk-forward recap (expanding vs sliding), embargo; regime intuition
- **(10 min)** Slides: defining regimes (rolling std), training-only thresholds, leakage pitfalls
- **(35 min) In-class lab**: add regime labels (train-only quantiles) → evaluate naive & linear-lags **by regime** → calibration plots
- **(10 min)** Wrap-up + homework brief
- **(10 min)** Buffer / Q&A

---

## 19.9 Slide talking points (paste into your deck)

### 19.9.1 Why regime analysis?

- Model error is **not uniform**. Many models fail during **high-volatility** periods.
- Reporting **one global metric** hides when/where models break.
- Regime-aware metrics guide **feature/model design** and **risk controls**.

### 19.9.2 Splits & embargo refresher

- **Rolling-origin, expanding**: train grows, validation moves forward.
- **Embargo**: gap (e.g., 5 business days) between train end and val start to reduce adjacency leakage.

### 19.9.3 Defining volatility regimes (avoid leakage)

- Use **rolling standard deviation** of returns (e.g., `roll_std_20`) computed **up to and including t**.
- **Thresholds**: choose quantiles (e.g., 33% and 66%) **on TRAIN ONLY** for each split; label both train & val using those fixed thresholds.
- **Categories**: `low`, `med`, `high`. Treat labels as **categorical dtypes**.

### 19.9.4 Metrics & calibration by regime

- Compute **MAE, sMAPE, MASE within each regime**. Aggregate macro/micro.
- **Calibration (point forecasts)**: bin predictions into deciles; plot **mean predicted vs. mean realized** per bin.
  - Perfect calibration points on the $45^\circ$ line.
  - Plot one figure **overall** and one **per regime**.

---

## 19.10 In-class lab (35 min, Colab-friendly)

Run each block as its **own cell**. Adjust `REPO_NAME` to your repo name.

### 19.10.1 0) Setup & load (with safe fallbacks)

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)

REPO_NAME  = "unified-stocks-teamX"   # <- change if needed
BASE_DIR   = "/content/drive/MyDrive/dspt25"
REPO_DIR   = f"{BASE_DIR}/{REPO_NAME}"

import os, pathlib, numpy as np, pandas as pd, json
from pathlib import Path
pathlib.Path(REPO_DIR).mkdir(parents=True, exist_ok=True)
os.chdir(REPO_DIR)
for p in ["data/raw","data/processed","data/static","reports","scripts","tests","docs/figs"]
    Path(p).mkdir(parents=True, exist_ok=True)
print("Working dir:", os.getcwd())

# Load returns; synthesize if missing
rpath = Path("data/processed/returns.parquet")
if rpath.exists():
    returns = pd.read_parquet(rpath)
else:
    rng = np.random.default_rng(0)
    dates = pd.bdate_range("2022-01-03", periods=360)
    frames=[]
    for t in ["AAPL","MSFT","GOOGL","AMZN","NVDA"]:
        eps = rng.normal(0,0.012,size=len(dates)).astype("float32")
        adj = 100*np.exp(np.cumsum(eps))
        df = pd.DataFrame({
            "date": dates, "ticker": t,
            "adj_close": adj.astype("float32"),
            "log_return": np.r_[np.nan, np.diff(np.log(adj))].astype("float32")
        })
        df["r_1d"] = df["log_return"].shift(-1)
        df["weekday"] = df["date"].dt.weekday.astype("int8")
        df["month"]   = df["date"].dt.month.astype("int8")
        frames.append(df)
    returns = pd.concat(frames, ignore_index=True).dropna().reset_index(drop=True)
    returns["ticker"] = returns["ticker"].astype("category")
    returns.to_parquet(rpath, index=False)

# Load features or generate minimal set with rolling std (causal)
```

```
fpath = Path("data/processed/features_v1.parquet")
if fpath.exists():
    feats = pd.read_parquet(fpath)
    if "roll_std_20" not in feats.columns:
        # ensure we have rolling volatility
        feats = feats.sort_values(["ticker","date"])
        feats["roll_std_20"] = feats.groupby("ticker")["log_return"].rolling(20, min_periods=
else:
    feats = returns.sort_values(["ticker","date"]).copy()
    for k in [1,2,3]:
        feats[f"lag{k}"] = feats.groupby("ticker")["log_return"].shift(k)
    feats["roll_std_20"] = feats.groupby("ticker")["log_return"].rolling(20, min_periods=20)

# If static universe exists from Session 17, apply it
univ_files = sorted(Path("data/static").glob("universe_*.csv"))
if univ_files:
    univ = pd.read_csv(univ_files[-1])["ticker"].astype(str)
    feats = feats[feats["ticker"].astype(str).isin(set(univ))]
    returns = returns[returns["ticker"].astype(str).isin(set(univ))]

# Harmonize types & sort
for df in (returns, feats):
    df["date"] = pd.to_datetime(df["date"])
    df["ticker"] = df["ticker"].astype("category")
feats = feats.dropna(subset=["log_return"]).sort_values(["ticker","date"]).reset_index(drop=
returns = returns.sort_values(["ticker","date"]).reset_index(drop=True)
feats.head(3)
```

### 19.10.2 1) Rolling-origin splits (expanding) with embargo

```
import numpy as np, pandas as pd

def make_rolling_origin_splits(dates, train_min=252, val_size=63, step=63, embargo=5):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    splits=[]; i=train_min-1; n=len(u)
    while True:
        if i>=n: break
        a,b = u[0], u[i]
        vs = i + embargo + 1
        ve = vs + val_size - 1
```

```
        if ve>=n: break
        splits.append((a,b,u[vs],u[ve]))
        i += step
    return splits

splits = make_rolling_origin_splits(feats["date"], train_min=252, val_size=63, step=63, emba
print("Num splits:", len(splits))
splits[:2]
```

### 19.10.3 2) Regime thresholds from training-only (quantiles of rolling vol)

```
def regime_thresholds(train_df, vol_col="roll_std_20", q_low=0.33, q_high=0.66):
    v = train_df[vol_col].dropna().to_numpy()
    if len(v) < 100:  # defensive: small train
        q_low, q_high = 0.4, 0.8
    return float(np.quantile(v, q_low)), float(np.quantile(v, q_high))

def label_regime(df, vol_col, lo, hi):
    # low: <= lo, high: >= hi, else med; NaNs -> 'unknown'
    out = df.copy()
    vc = out[vol_col]
    regime = pd.Series(pd.Categorical(["unknown"]*len(out), categories=["low","med","high","
    regime[(vc.notna()) & (vc <= lo)] = "low"
    regime[(vc.notna()) & (vc > lo) & (vc < hi)] = "med"
    regime[(vc.notna()) & (vc >= hi)] = "high"
    out["regime"] = regime.astype("category")
    return out

# Demonstrate on first split in class
a,b,c,d = splits[0]
tr = feats[(feats["date"]>=a) & (feats["date"]<=b)]
va = feats[(feats["date"]>=c) & (feats["date"]<=d)]
lo, hi = regime_thresholds(tr, "roll_std_20", 0.33, 0.66)
tr_lab = label_regime(tr, "roll_std_20", lo, hi)
va_lab = label_regime(va, "roll_std_20", lo, hi)
print({"lo": lo, "hi": hi}, tr_lab["regime"].value_counts().to_dict(), va_lab["regime"].value
```

### 19.10.4 3) Baseline predictions (naive & linear-lags per ticker, fit on TRAIN only)

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

# features we will use for linear baseline
XCOLS = [c for c in ["lag1","lag2","lag3"] if c in feats.columns]
if not XCOLS:
    # create lags on the fly (causal)
    feats = feats.sort_values(["ticker","date"]).copy()
    for k in [1,2,3]:
        feats[f"lag{k}"] = feats.groupby("ticker")["log_return"].shift(k)
    XCOLS = ["lag1","lag2","lag3"]

def fit_predict_lin_per_ticker(train_df, val_df):
    preds=[]
    for tkr, trk in train_df.groupby("ticker"):
        vak = val_df[val_df["ticker"]==tkr]
        if len(trk)==0 or len(vak)==0: continue
        pipe = Pipeline([("scaler", StandardScaler()), ("lr", LinearRegression())])
        pipe.fit(trk[XCOLS].dropna().values, trk.dropna(subset=XCOLS)["r_1d"].values)
        yhat = pipe.predict(vak[XCOLS].fillna(0).values)
        out = vak[["date","ticker","r_1d","log_return","regime"]].copy()
        out["yhat_lin"] = yhat.astype("float32")
        preds.append(out)
    return pd.concat(preds, ignore_index=True) if preds else pd.DataFrame()

def add_naive_preds(df):
    out = df.copy()
    out["yhat_naive"] = out["log_return"]  # r_{t+1} ~ log_return_t
    return out

tr_lab2 = add_naive_preds(tr_lab)
va_lab2 = add_naive_preds(va_lab)
val_lin = fit_predict_lin_per_ticker(tr_lab2, va_lab2)
val = va_lab2.merge(val_lin[["date","ticker","yhat_lin"]], on=["date","ticker"], how="left")
val.head(3)
```

### 19.10.5 4) Metrics by regime (MAE, sMAPE, MASE; macro & micro)

```
def mae(y, yhat):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(np.abs(y - yhat)))

def smape(y,yhat,eps=1e-8):
    y = np.asarray(y); yhat = np.asarray(yhat)
    return float(np.mean(2.0*np.abs(y-yhat)/(np.abs(y)+np.abs(yhat)+eps)))

def mase(y_true, y_pred, y_train_true, y_train_naive):
    scale = mae(y_train_true, y_train_naive) + 1e-12
    return float(mae(y_true,y_pred)/scale)

def per_regime_metrics(val_df, train_df, pred_col):
    rows=[]
    for reg, g in val_df.groupby("regime"):
        if reg == "unknown" or len(g)==0:
            continue
        # build per-ticker MASE scales from TRAIN
        per_t = []
        for tkr, gv in g.groupby("ticker"):
            gt = train_df[train_df["ticker"]==tkr].dropna(subset=["r_1d"])
            if len(gt)==0: continue
            m = {
                "ticker": tkr,
                "n": int(gv["r_1d"].notna().sum()),
                "mae": mae(gv["r_1d"], gv[pred_col]),
                "smape": smape(gv["r_1d"], gv[pred_col]),
                "mase": mase(gv["r_1d"], gv[pred_col], gt["r_1d"], gt["log_return"]),
                "regime": reg
            }
            per_t.append(m)
        per_t = pd.DataFrame(per_t)
        if per_t.empty:
            continue
        # macro (mean of per-ticker)
        macro = per_t[["mae","smape","mase"]].mean().to_dict()
        # micro (weighted by n)
        w = per_t["n"].to_numpy()
        micro = {
            "micro_mae": float(np.average(per_t["mae"], weights=w)),
```

```
            "micro_smape": float(np.average(per_t["smape"], weights=w)),
            "micro_mase": float(np.average(per_t["mase"], weights=w)),
        }
        rows.append({"regime":reg, **{f"macro_{k}":float(v) for k,v in macro.items()}, **micr
    return pd.DataFrame(rows)

met_naive = per_regime_metrics(val, tr_lab2, "yhat_naive")
met_lin   = per_regime_metrics(val.dropna(subset=["yhat_lin"]), tr_lab2, "yhat_lin")
print("NAIVE by regime:\n", met_naive)
print("\nLIN-LAGS by regime:\n", met_lin)
# Save
pd.concat([
    met_naive.assign(model="naive"),
    met_lin.assign(model="lin_lags")
], ignore_index=True).to_csv("reports/regime_metrics_split1.csv", index=False)
```

### 19.10.6 5) Calibration plots overall and by regime (binned)

```
import matplotlib.pyplot as plt
import numpy as np, pandas as pd, pathlib

def calibration_by_bins(df, pred_col, y_col="r_1d", n_bins=10):
    d = df.dropna(subset=[pred_col, y_col]).copy()
    d["bin"] = pd.qcut(d[pred_col], q=n_bins, duplicates="drop")
    grp = d.groupby("bin").agg(
        mean_pred=(pred_col, "mean"),
        mean_true=(y_col, "mean"),
        count=(y_col, "size")
    ).reset_index()
    return grp

# Overall calibration (lin_lags) on validation slice
cal_overall = calibration_by_bins(val.dropna(subset=["yhat_lin"]), "yhat_lin", "r_1d", n_bins

plt.figure(figsize=(5,4))
plt.plot(cal_overall["mean_pred"], cal_overall["mean_true"], marker="o")
lim = max(abs(cal_overall["mean_pred"]).max(), abs(cal_overall["mean_true"]).max())
plt.plot([-lim, lim], [-lim, lim], linestyle="--")
plt.xlabel("Mean predicted (bin)"); plt.ylabel("Mean realized (bin)")
plt.title("Calibration (overall) - lin_lags")
```

```
plt.tight_layout()
plt.savefig("docs/figs/calibration_overall_lin.png", dpi=160)
"Saved docs/figs/calibration_overall_lin.png"

# By regime
plt.figure(figsize=(6.5,4.5))
for i, reg in enumerate(["low","med","high"], start=1):
    g = val[(val["regime"]==reg) & (val["yhat_lin"].notna())]
    if len(g) < 50:
        continue
    cal = calibration_by_bins(g, "yhat_lin", "r_1d", n_bins=6)
    plt.plot(cal["mean_pred"], cal["mean_true"], marker="o", label=reg)
lim = 0.02  # small returns
plt.plot([-lim, lim], [-lim, lim], linestyle="--")
plt.xlabel("Mean predicted (bin)"); plt.ylabel("Mean realized (bin)")
plt.title("Calibration by regime - lin_lags")
plt.legend()
plt.tight_layout()
plt.savefig("docs/figs/calibration_by_regime_lin.png", dpi=160)
"Saved docs/figs/calibration_by_regime_lin.png"
```

---

## 19.11 Wrap-up (10 min) — key points to emphasize

- **Regime thresholds must be set on TRAIN ONLY** each split to avoid leakage.
- Report **by-regime** metrics alongside overall metrics; show **macro** & **micro**.
- Calibration plots (binned predicted vs. realized) quickly show **systematic bias**; compare regimes.

---

## 19.12 Homework (due before Session 19)

**Goal:** Produce a **full regime-aware evaluation** across **all splits** for **naive** and **linear-lags** models and include the figures in your Quarto report.

### 19.12.1 A. Script: `scripts/regime_eval.py` — run across all splits

```
#!/usr/bin/env python
from __future__ import annotations
import argparse, json, numpy as np, pandas as pd
from pathlib import Path
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression

def make_splits(dates, train_min=252, val_size=63, step=63, embargo=5):
    u = np.array(sorted(pd.to_datetime(pd.Series(dates).unique())))
    splits=[]; i=train_min-1; n=len(u)
    while True:
        if i>=n: break
        a,b = u[0], u[i]; vs=i+embargo+1; ve=vs+val_size-1
        if ve>=n: break
        splits.append((a,b,u[vs],u[ve])); i+=step
    return splits

def regime_thresholds(train_df, vol_col="roll_std_20", q_low=0.33, q_high=0.66):
    v = train_df[vol_col].dropna().to_numpy()
    if len(v) < 100:
        q_low, q_high = 0.4, 0.8
    return float(np.quantile(v, q_low)), float(np.quantile(v, q_high))

def label_regime(df, vol_col, lo, hi):
    out = df.copy()
    vc = out[vol_col]
    reg = pd.Series(pd.Categorical(["unknown"]*len(out), categories=["low","med","high","unk
    reg[(vc.notna()) & (vc <= lo)] = "low"
    reg[(vc.notna()) & (vc > lo) & (vc < hi)] = "med"
    reg[(vc.notna()) & (vc >= hi)] = "high"
    out["regime"] = reg.astype("category")
    return out

def add_naive(df):
    out = df.copy()
    out["yhat_naive"] = out["log_return"]
    return out

def fit_lin(tr, va, xcols):
```

```
    from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import StandardScaler
    from sklearn.linear_model import LinearRegression
    preds=[]
    for tkr, trk in tr.groupby("ticker"):
        vak = va[va["ticker"]==tkr]
        if len(trk)==0 or len(vak)==0: continue
        Xtr = trk.dropna(subset=xcols);
        pipe = Pipeline([("scaler", StandardScaler()), ("lr", LinearRegression())])
        pipe.fit(Xtr[xcols].values, Xtr["r_1d"].values)
        yhat = pipe.predict(vak[xcols].fillna(0).values)
        out = vak[["date","ticker","r_1d","log_return","regime"]].copy()
        out["yhat_lin"] = yhat
        preds.append(out)
    return pd.concat(preds, ignore_index=True) if preds else pd.DataFrame()

def mae(y, yhat): y=np.asarray(y); yhat=np.asarray(yhat); return float(np.mean(np.abs(y-yhat
def smape(y,yhat,eps=1e-8):
    y=np.asarray(y); yhat=np.asarray(yhat); return float(np.mean(2*np.abs(y-yhat)/(np.abs(y)
def mase(y_true, y_pred, y_train_true, y_train_naive):
    return float(mae(y_true, y_pred)/(mae(y_train_true, y_train_naive)+1e-12))

def per_regime_metrics(val_df, train_df, pred_col):
    rows=[]
    for reg, g in val_df.groupby("regime"):
        if reg=="unknown" or len(g)==0: continue
        per=[]
        for tkr, gv in g.groupby("ticker"):
            gt = train_df[train_df["ticker"]==tkr].dropna(subset=["r_1d"])
            if len(gt)==0: continue
            per.append({"ticker":tkr,"n":int(gv["r_1d"].notna().sum()),
                        "mae": mae(gv["r_1d"], gv[pred_col]),
                        "smape": smape(gv["r_1d"], gv[pred_col]),
                        "mase": mase(gv["r_1d"], gv[pred_col], gt["r_1d"], gt["log_return"])
                        "regime": reg})
        pt = pd.DataFrame(per)
        if pt.empty: continue
        macro = pt[["mae","smape","mase"]].mean().to_dict()
        w = pt["n"].to_numpy()
        micro = {"micro_mae": float(np.average(pt["mae"], weights=w)),
                 "micro_smape": float(np.average(pt["smape"], weights=w)),
                 "micro_mase": float(np.average(pt["mase"], weights=w))}
```

```
        rows.append({"regime":reg, **{f"macro_{k}":float(v) for k,v in macro.items()}, **mic
    return pd.DataFrame(rows)

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--features", default="data/processed/features_v1.parquet")
    ap.add_argument("--train-min", type=int, default=252)
    ap.add_argument("--val-size", type=int, default=63)
    ap.add_argument("--step", type=int, default=63)
    ap.add_argument("--embargo", type=int, default=5)
    ap.add_argument("--vol-col", default="roll_std_20")
    ap.add_argument("--xcols", nargs="+", default=["lag1","lag2","lag3"])
    ap.add_argument("--out-summary", default="reports/regime_summary.csv")
    args = ap.parse_args()

    df = pd.read_parquet(args.features).sort_values(["ticker","date"]).reset_index(drop=True
    # Ensure vol col exists
    if args.vol_col not in df.columns:
        df[args.vol_col] = df.groupby("ticker")["log_return"].rolling(20, min_periods=20).st

    # Build lags if missing
    for k in [1,2,3]:
        col = f"lag{k}"
        if col not in df.columns:
            df[col] = df.groupby("ticker")["log_return"].shift(k)

    splits = make_splits(df["date"], args.train_min, args.val_size, args.step, args.embargo)
    Path("reports").mkdir(parents=True, exist_ok=True)
    thresh_rec = {}

    rows=[]
    for sid,(a,b,c,d) in enumerate(splits, start=1):
        tr = df[(df["date"]>=a)&(df["date"]<=b)]
        va = df[(df["date"]>=c)&(df["date"]<=d)]
        lo, hi = regime_thresholds(tr, args.vol_col)
        thresh_rec[sid] = {"lo":lo, "hi":hi, "train_range":f"{a.date()}→{b.date()}"}
        trL = label_regime(tr, args.vol_col, lo, hi)
        vaL = label_regime(va, args.vol_col, lo, hi)

        # predictions
        trN, vaN = add_naive(trL), add_naive(vaL)
        val_lin = fit_lin(trN, vaN, args.xcols)
```

```python
        vaN = vaN.merge(val_lin[["date","ticker","yhat_lin"]], on=["date","ticker"], how="le

        # metrics
        m_naive = per_regime_metrics(vaN, trN, "yhat_naive").assign(split=sid, model="naive"
        m_lin   = per_regime_metrics(vaN.dropna(subset=["yhat_lin"]), trN, "yhat_lin").assig

        out = pd.concat([m_naive, m_lin], ignore_index=True)
        out.to_csv(f"reports/regime_metrics_split{sid}.csv", index=False)
        rows.append(out)

    pd.concat(rows, ignore_index=True).to_csv(args.out_summary, index=False)
    Path("reports/regime_thresholds.json").write_text(json.dumps(thresh_rec, indent=2))
    print("Wrote", args.out_summary, "and per-split CSVs; thresholds saved to reports/regime_

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

Run:

```bash
%%bash
chmod +x scripts/regime_eval.py
python scripts/regime_eval.py
```

### 19.12.2 B. Plot summary figures for your report

```python
import pandas as pd, matplotlib.pyplot as plt, pathlib
pathlib.Path("docs/figs").mkdir(parents=True, exist_ok=True)

df = pd.read_csv("reports/regime_summary.csv")
# Micro MAE by regime per model
pivot = df.pivot_table(index=["split","regime"], columns="model", values="micro_mae")
plt.figure(figsize=(6,4))
for model in pivot.columns:
    plt.plot(pivot.xs("low", level="regime").index, pivot.xs("low", level="regime")[model], m
    plt.plot(pivot.xs("high", level="regime").index, pivot.xs("high", level="regime")[model]
plt.xlabel("Split"); plt.ylabel("Micro MAE")
plt.title("Micro MAE by regime (low vs high)")
plt.legend(); plt.tight_layout()
plt.savefig("docs/figs/regime_micro_mae.png", dpi=160)
"Saved docs/figs/regime_micro_mae.png"
```