1 Collaborating on Your Final Project

You've been using Git and GitHub all semester to submit homework - add, commit, push, pull. For the final project, you'll use those same commands, but now with a partner. This introduces a new challenge: two people editing the same repository.

This guide covers how to set up your shared project and avoid the most common collaboration headaches. It will cover:

1. Why collaboration is hard
2. How to address that strategically
3. Using git / GitHub to your advantage (optional)

1.1 Why Collaboration Gets Complicated

When you work alone, your local copy and GitHub stay in sync easily. With two people, a new problem emerges: you both pull the latest version, you both make changes, and then you both try to push. The second person to push gets rejected - Git doesn't know how to combine your changes automatically.

This is called a **merge conflict**. With regular Python files, Git can often resolve minor conflicts on its own - if you edited lines 10-20 and your partner edited lines 50-60, Git merges both changes without complaint. But when Git can't figure it out, it marks the conflicting sections and asks you to resolve them manually.

With Jupyter notebooks, merge conflicts are particularly painful. To understand why, you need to know how notebooks are actually stored.

1.1.1 What's Inside a Notebook File

Notebooks are stored as JSON (JavaScript Object Notation) - a text format that looks like nested Python dictionaries. When you save a notebook, Jupyter doesn't save something human-readable. It saves something like this:

```json
{
  "cells": [
    {
      "cell_type": "code",
      "execution_count": 42,
      "outputs": [{ "data": { "text/plain": ["0    1.0\n", "1    2.0\n"] } }],
      "source": ["import pandas as pd\n", "pd.Series([1, 2, None])"]
    }
  ],
  "metadata": { "kernelspec": { "display_name": "Python 3" } },
  "nbformat": 4
}
```

Notice that the file contains not just your code (`"source"`), but also the output, the execution count (how many cells you've run), and various metadata. Two people can run identical code and produce different files because their execution counts differ.

When a merge conflict happens in a `.py` file, you can read it and fix it. When a merge conflict happens in a notebook, the resulting JSON is often broken - Jupyter can't even open it. You're left staring at corrupted JSON trying to figure out what went wrong.

1.2 Mitigating Merge Problems

We can mitigate this in two ways:

1. Divide ownership so you're rarely editing the same notebook
2. Eliminate meaningless differences (like outputs and formatting) that create unnecessary conflicts

1.2.1 Divide Your Work

The single most effective way to avoid conflicts: don't edit the same files at the same time.

Before you start coding, have a conversation about how to split the work. There are two common approaches:

- Option 1: Divide by file. Each partner owns specific notebooks. On any given day, you're not both editing the same file.
- Option 2: Divide by time (rotating ownership). You rotate - one person works on the project Monday/Wednesday, the other works Tuesday/Thursday. Whoever is "off" pulls the latest before their next session.

Either can work. The key is agreeing on a system and communicating.

Note that EDA work is often iterative and non-linear. You might discover something in exploration that sends you back to data cleaning. Dividing work strictly by phase (cleaning → exploration → modeling) can be challenging in practice. Be flexible and communicate when boundaries shift.

Doing this effectively requires good communication. Use text messages or whatever you normally use to communicate with your partner. Don't work on the same file at the same time.

1.2.1.1 Coordinate Your Interfaces

However you divide the work, you'll need to agree on how notebooks connect to each other. If Partner A's cleaning notebook produces a file that Partner B's exploration notebook reads, agree on:

- The filename (`cleaned_data.csv`)
- The location (`data/` folder)
- The structure (what columns exist, what the index is)

1.2.1.2 Pickle Your Data

When saving intermediate DataFrames for your partner to use, consider using pickle format instead of CSV. It is a trivial change:

```
# Saving (in cleaning notebook)
df.to_pickle("../data/cleaned_data.pkl")

# Loading (in exploration notebook)
df = pd.read_pickle("../data/cleaned_data.pkl")
```

CSV files are plain text and do not retain type information. Pickle preserves the DataFrame exactly: dtypes, index, datetime columns, categorical encodings, everything. What you save is what you get back.

1.2.1.3 Use Shared Utility Code (Optional)

If you find yourselves copying the same helper functions between notebooks, consider putting them in a shared `.py` file. Python files produce readable diffs and merge more gracefully than notebooks.

For example, you might create a `utils.py` file (in the same directory as the ipynb):

```
# utils.py
import pandas as pd

def load_clean_data(path="../data/cleaned_data.csv"):
    """Load the cleaned dataset with standard preprocessing."""
    df = pd.read_pickle(path)
    df['date'] = pd.to_datetime(df['date'])
    return df
```

Then in your notebooks, import and use it:

```
from utils import load_clean_data

df = load_clean_data()
```

This pattern should look familiar from the Streamlit lecture, where we imported functions from separate modules. It is entirely optional - for a semester project, some duplication between notebooks is fine.

1.2.2 Eliminate Meaningless Differences

Clear outputs manually before committing any change. In Jupyter, use Cell → All Output → Clear or Kernel → Restart Kernel and Clear All Outputs before saving. This eliminates the biggest source of spurious conflicts, as described above.

1.3 A Better Way?

The workflow described above represents best practices for collaborative data science work. It will reduce friction and help you avoid painful merge conflicts. But it is mostly manual, somewhat fragile, and does not leverage the tools you have. It assumes you are either sharing code manually and/or collaborating synchronously (in person or otherwise). That approach is completely acceptable for this project, but there is a "better" way...

The rest of this guide assumes you want a more automated path, similar to what is used by professionals, and are willing to invest a little in learning the process and tools required. It is optional.

If you want to try this approach but have already begun work on the project, see Appendix A for migration steps. But read the rest first.

1.3.1 Setting Up the Shared Repository

You'll create a **new repository** for this project, separate from your homework repo. This keeps the project self-contained as a portfolio piece for both partners. It also avoids a privacy issue - if you added your partner as a collaborator on your homework repo, they'd have access to all your past submissions.

After setup, your folder structure will look like this:

```
~/insy6500/
    my_repo/        → your homework repo on GitHub
    class_repo/     → clone of course content
    my_project/     → clone of shared project repo (NEW)
```

Both partners will have a `my_project` folder that points to the same GitHub repository.

1.3.1.1 Partner A: Create the Repository

Follow the same steps you used to create `my_repo` in HW3A:

```
cd ~/insy6500
mkdir my_project
cd my_project
git init
```

Create a README file so the repository isn't empty:

```
echo "# EDA Project" > README.md
git add README.md
git commit -m "Initial commit"
```

Now create the repository on GitHub and connect it, just like you did in HW3A (Task 4: Publish to GitHub):

1. Go to github.com → click "+" → "New repository"
2. Name it something meaningful (e.g., `eda-project`)
3. Add a brief description, leave other settings as defaults
4. Click "Create repository"

Back in your terminal, connect and push (replace `<USERNAME>` with your GitHub username):

```
git remote add origin https://github.com/<USERNAME>/eda-project.git
git push -u origin main
```

Once complete, go to your repository on GitHub:

1. Click Settings → Collaborators → Add people
2. Enter Partner B's GitHub username and send the invitation

1.3.1.2 Partner B: Accept and Clone

1. Check your email or GitHub notifications for the invitation
2. Accept the invitation
3. Clone the repository:

   ```
   cd ~/insy6500
   git clone https://github.com/partner-a-username/eda-project.git my_project
   ```

Now you both have push access to the same repository. Your `my_project` folders point to the same place on GitHub.

1.3.1.3 Verify Both Partners Have Access

Before going further, confirm that both partners can push and pull.

Partner B (since you just cloned, test your push access):

Copy the `.gitignore` from your homework repo and push it:

```
cd ~/insy6500/my_project
cp ~/insy6500/my_repo/.gitignore .
git add .gitignore
git commit -m "Add .gitignore"
git push
```

Partner A (verify you can pull Partner B's changes):

```
cd ~/insy6500/my_project
git pull
```

You should see the `.gitignore` file appear. If both steps succeed, you're ready to proceed.

1.3.2 Setting Up Your Tools

Both partners must complete these steps. Make sure your class environment is activated first:

```
conda activate insy6500
```

1.3.2.1 Install nbstripout

Jupyter notebooks store their output (tables, plots, printed results) in the file itself, along with transient metadata like execution counts. This causes problems:

- Two people run the same cell, get identical output, but the execution count differs - Git sees this as a conflict
- Large outputs bloat the repository
- Diffs become unreadable (you can't see code changes through walls of output)

The tool `nbstripout` solves this by automatically stripping output and other cruft when you stage files with `git add`.

```
conda install nbstripout
```

Then, inside your project directory:

```
cd ~/insy6500/my_project
nbstripout --install
```

This configures Git to run nbstripout automatically during `git add`. Your working notebook still shows output - you just don't commit it.

! Important

This only works if your class environment is activated when you run git commands. If nbstripout isn't on your PATH, the filter silently fails and you commit outputs anyway. Always activate your environment before doing any git operations.

1.3.2.2 Install and Configure ruff

Formatting differences create merge conflicts too. If one partner uses single quotes and the other uses double quotes, or you have different habits around spacing, Git sees every reformatted line as a change.

`ruff` is a fast Python formatter that handles notebooks directly. It automatically reformats code to a consistent style.

```
conda install ruff
```

One partner (either A or B) should create a `ruff.toml` file in the `my_project` folder. Do this before you start adding notebooks, so both partners have the same settings from the beginning:

```
cd ~/insy6500/my_project
```

Create `ruff.toml` (it's just a text file, use nano) with these contents:

```
line-length = 100
indent-width = 4

[format]
quote-style = "double"
indent-style = "space"
```

Then commit and push:

```
git add ruff.toml
git commit -m "Add ruff configuration"
git push
```

The other partner pulls to get the config file:

```
git pull
```

Now both partners have identical formatting settings. Before committing, format your code:

```
ruff format .
```

This reformats all Python files and notebooks in the current directory. Run this every time before you commit - it takes less than a second to go from something like this:

```
def calculate_summary_stats(df,column,include_median=True):
  result={'count':len(df),
    'mean':df[column].mean(),
      'std':df[column].std()}
  if include_median==True:
        result['median']=df[column].median()
  return result
```

To this:

```
def calculate_summary_stats(df, column, include_median=True):
    result = {
        "count": len(df),
        "mean": df[column].mean(),
        "std": df[column].std(),
    }
    if include_median == True:
        result["median"] = df[column].median()
    return result
```

I feel better already! Because this process is deterministic, ruff eliminates personal styling habits and guarantees that git only identifies meaningful changes in the code. Highly recommended.

1.3.3 The Daily Workflow

This is the same pull-commit-push cycle you've used all semester, with one addition: formatting before you commit.

1.3.3.1 Before You Start Working

Always pull the latest changes before you start:

```
cd ~/insy6500/my_project
git pull
```

This gets your partner's recent commits. If you skip this and you've both made changes, you'll have to merge - which is what we're trying to avoid.

1.3.3.2 When You're Ready to Commit

1. Format your code:

   ```
   ruff format .
   ```

2. Check what's changed:

   ```
   git status
   ```

3. Stage, commit, push:

   ```
   git add .
   git commit -m "Add correlation analysis to exploration notebook"
   git push
   ```

The nbstripout filter runs automatically during `git add`, stripping outputs from notebooks before they're staged.

1.3.3.3 If Push Fails

If you see this:

```
! [rejected]        main -> main (fetch first)
error: failed to push some refs to '...'
```

It means your partner pushed changes while you were working. Run:

```
git pull
```

Git will attempt to merge your partner's changes with yours. If you edited different files (or different parts of the same `.py` file), Git usually handles this automatically and you can then push.

If Git can't resolve the differences, you'll get a merge conflict...

1.3.4 When Things Go Wrong

1.3.4.1 Merge Conflicts in Notebooks

If you see a conflict in a `.ipynb` file, **don't try to edit the JSON by hand**.

For simple conflicts (e.g., you forgot to run `ruff format .` before committing), you may be able to resolve it quickly on your own. But for anything non-trivial - if you're unsure whose code is "right" or how to combine changes - get on a Zoom/Teams call with your partner (or meet in person) and work through it together with screen sharing. Two sets of eyes make this much easier, and you'll learn more by talking through it.

You have two options...

Option 1: Keep one version

Decide whose version of the conflicting file(s) to keep. Take careful note of changes made in the other version, if you need to reproduce them. Then use `git checkout` to restore the best version:

```
# Keep YOUR version, discard partner's changes to this file
git checkout --ours notebook.ipynb
git add notebook.ipynb
```

```
# OR keep PARTNER'S version, discard your changes to this file
git checkout --theirs notebook.ipynb
git add notebook.ipynb
```

Edit that file to reapply any additional changes, then complete the merge:

```
git commit -m "Resolve conflict, keeping [my/partner's] version"
```

The resulting commit should include the "best" version (ours / theirs) plus the changes you manually reproduced.

Option 2: Abort and regroup

If you're confused about what's happening, you can abort the merge entirely:

```
git merge --abort
```

This puts you back to where you were before the `git pull`. Coordinate with your partner and figure out a plan before trying again.

1.3.4.2 Merge Conflicts in .py Files

These are more manageable. Git often resolves minor conflicts automatically - if you edited lines 10-20 and your partner edited lines 50-60, Git merges both changes without complaint.

When Git can't figure it out, it marks the conflict in the file:

```
<<<<<<< HEAD
def clean_data(df):
    return df.dropna()
=======
def clean_data(df):
    return df.dropna(subset=['important_column'])
>>>>>>> abc1234
```

Everything between `<<<<<<< HEAD` and `=======` is your version. Everything between `=======` and `>>>>>>> abc1234` is your partner's version. Edit the file to combine the changes correctly (or pick one), remove all the conflict markers, then:

```
git add utils.py
git commit -m "Resolve merge conflict in clean_data function"
```

1.3.4.3 "I'm Stuck and Nothing Makes Sense"

Git errors can be cryptic. ChatGPT, Claude, and the like are excellent at interpreting git error messages and suggesting solutions.

When asking for help:

- Paste the error message (the exact text, not a paraphrase)
- Describe what you were trying to do and what happened leading up to the error
- Include context: the output of `git status` is often helpful

For really tangled situations, you can even paste a large chunk of your terminal history. Run the `history` command to see recent commands, or just copy-paste the last 50-100 lines from your terminal window. LLMs can handle hundreds of lines of context and often spot the problem faster than you can explain it.

1.3.4.4 Before the Nuclear Option: Get Help

If you feel like everything is broken, pause and ask for help before doing anything drastic. Reach out to the teaching team or ask ChatGPT/Claude to diagnose the situation. What looks unfixable is often recoverable with the right command - and you'll learn something in the process.

1.3.4.5 The Nuclear Option

If everything is truly broken, you've consulted the teaching team or an LLM, and there's no clear path forward, remember: **you both have the work locally**. The worst-case scenario is:

1. Copy the contents of your `my_project` folder somewhere safe, just in case!
2. Delete the repository on GitHub
3. Delete your local repo: `rm -rf ~/insy6500/my_project/.git`
4. Start fresh, following the process outlined above:
   - Re-initialize the repo with git init
   - Add everything to it with git add and commit
   - Create the GitHub repo again
   - Set up the remote and push the contents
   - Add your partner as a collaborator

All you lose is the commit history. The actual work still exists on both of your machines. This is the "break glass in emergency" option - rarely needed, but reassuring to know it's there.

1.3.5 Quick Reference

Start of each work session:

```
conda activate insy6500
cd ~/insy6500/my_project
git pull
```

Before each commit:

```
ruff format .
git status
git add .
git commit -m "Descriptive message"
git push
```

Tools to install (both partners):

```
conda activate insy6500
conda install nbstripout ruff
cd ~/insy6500/my_project
nbstripout --install
```

Golden rules:

1. Always pull before you start working
2. Coordinate ownership - avoid editing the same notebooks at the same time
3. Activate your environment before git commands
4. Format with `ruff format .` before you commit
5. Communicate with your partner

1.4 Appendix A: Migrating an Existing Project

If you started the project before reading this guide, here's how to migrate to the recommended structure.

1.4.1 Scenario 1: You've Been Working in Your Homework Repo

If you started the project inside `my_repo` (your homework repository):

1. Copy your project files somewhere temporary:

   ```
   mkdir ~/temp_project
   cp ~/insy6500/my_repo/project/*.ipynb ~/temp_project/
   cp ~/insy6500/my_repo/project/data/* ~/temp_project/  # if applicable
   ```

2. Follow the "Setting Up the Shared Repository" steps in this guide to create `my_project` (same process as HW3A)

3. Copy your files into the new repo:

   ```
   cp ~/temp_project/*.ipynb ~/insy6500/my_project/
   ```

4. Set up nbstripout and ruff as described in this guide

5. Commit and push:

   ```
   cd ~/insy6500/my_project
   ruff format .
   git add .
   git commit -m "Migrate project files"
   git push
   ```

6. Have your partner clone the new repo

1.4.2 Scenario 2: You Already Have a Shared Repo Without the Tools

If you've already set up a shared repo but haven't installed nbstripout or ruff:

1. Both partners: Install the tools as described in "Setting Up Your Tools"

2. One partner: Create and push `ruff.toml`

3. Other partner: Pull to get the config
4. Both partners: Clear notebook outputs manually one time:
   - Open each notebook in Jupyter
   - *Cell → All Output → Clear*
   - Save the notebook
5. One partner: Commit the cleared notebooks:

```
ruff format .
git add .
git commit -m "Clear outputs, standardize formatting"
git push
```

6. Other partner: Pull the cleaned versions

From this point forward, nbstripout will handle outputs automatically.

1.5 Appendix B: Adding the Project to Partner B's GitHub (After the Semester)

The project repository lives under Partner A's GitHub account. Partner B is a collaborator with full access, and their commits show up in their GitHub contribution history, but the repo doesn't appear in their own repository list.

After the semester is complete - or any time after the project is finished - Partner B can create a copy under their own account for portfolio purposes by *forking* the repository. Forking creates your own independent copy of someone else's repository on GitHub. Unlike cloning (which downloads to your computer), a fork lives on GitHub under your account. You own it completely, can modify it without affecting the original, and it appears in your public profile as your repository.

To do that:

1. Go to the project repo on GitHub (under Partner A's account)
2. Click the Fork button in the upper right
3. This creates a copy under Partner B's account

The fork is a snapshot - future changes to Partner A's repo won't automatically appear in the fork, and vice versa. But for a completed project, that's fine. Both partners now have the project visible on their own GitHub profiles.

Note

This is only relevant after you're done collaborating. During the project, both partners work on the same repo - don't fork it while you're still making changes.