Git Fundamentals for Data Science Team Collaboration

Learning Objectives

By the end of this session, you will:

- Understand Git's role in collaborative data science projects
- Master essential Git workflows for team environments
- Handle basic merge conflicts
- Create simple branching strategies
- Write professional commit messages
- Use basic collaboration workflows

Prerequisites

- Basic command line familiarity
- Understanding of file systems and directories
- Experience with any programming language (Python preferred)

---

1. Why Git Matters in Data Science Teams

The Problem Without Version Control

- Multiple people working on the same code
- Lost work when files are overwritten
- No way to track who made what changes
- Difficulty rolling back to working versions

Git Provides

1. **Complete History**: Every change is tracked and reversible
2. **Parallel Development**: Multiple people can work simultaneously
3. **Conflict Resolution**: Smart merging when changes overlap
4. **Backup**: Your work exists in multiple places
5. **Accountability**: Clear record of who changed what and when

```
graph TD
    A[Team Project] --> B[Sarah: Data Pipeline]
    A --> C[Mike: Model Experiments]
    A --> D[You: Evaluation Metrics]

    B --> E[Git Repository]
    C --> E
    D --> E

    E --> F[Integrated Solution]
    E --> G[Version History]
    E --> H[Backup Safety]

    F --> I[Production Deployment]
```

---

2. Git's Three-Stage Workflow

Git operates on a three-stage system:

```
graph LR
    A[Working Directory] -->|git add| B[Staging Area]
    B -->|git commit| C[Repository]
    C -->|git push| D[Remote Repository]

    D -->|git pull| C
    C -->|git checkout| A
```

Think of it like preparing a research publication:

- **Working Directory**: Your messy draft with experiments
- **Staging Area**: Carefully selected sections ready for review
- **Repository**: The published version with documentation
- **Remote Repository**: Shared location where everyone can access it

Step 1: Working Directory - Your Playground

This is where you make all your changes - writing code, creating notebooks, modifying data files.

```
# Check what's in your working directory
git status

# See which files have changes
git diff
```

Key Point: Your working directory is like your lab bench - messy, experimental, full of works-in-progress.

Step 2: Staging Area - Quality Control

The staging area lets you prepare exactly what you want to commit, creating clean, logical commits.

```
# Add specific files to staging
git add data_preprocessing.py
git add models/random_forest.py

# Add all modified files
git add .

# See what's staged
git status
git diff --cached  # See staged changes
```

Key Point: You can stage only parts of your work, leaving experiments for later. This creates clean commits instead of messy snapshots.

Basic Commands

```
# Check status
git status

# Stage files
git add filename.py
git add .  # Add all files

# Commit changes
git commit -m "Add data preprocessing pipeline"

# Push to remote
git push origin main

# Pull latest changes
git pull origin main
```

---

3. Writing Good Commit Messages

Format

```
Short summary (50 characters or less)

Optional detailed explanation:
- What problem does this solve?
- How does it solve it?
```

Examples

Bad:

```
fix stuff
update model
WIP
```

Good:

```
Fix memory leak in data preprocessing

The pandas operations were not releasing memory after
processing large CSV files. Switch to chunked reading
for files larger than 1GB.
```

```
Add hyperparameter tuning for RandomForest

Implement GridSearchCV with cross-validation.
Results in 12% accuracy improvement on validation set.
```

---

4. Basic Branching

Why Use Branches?

Branches allow multiple developers to work on different features without interfering with each other.

```
graph TD
    A[main branch] --> B[feature/data-pipeline]
    A --> C[feature/new-model]
    A --> D[bugfix/critical-issue]

    B --> E[Merge to main]
    C --> F[Merge to main]
    D --> G[Merge to main]

    E --> H[main branch updated]
    F --> H
    G --> H
```

Basic Branch Commands

```
# See all branches
git branch

# Create and switch to new branch
git checkout -b feature/model-optimization

# Switch between branches
git checkout main
git checkout feature/model-optimization

# Merge branch into main
git checkout main
git merge feature/model-optimization

# Delete branch after merging
git branch -d feature/model-optimization
```

Branch Naming Convention

```
feature/short-description    # New functionality
bugfix/issue-description     # Bug fixes
experiment/model-name        # Research experiments
```

---

5. Handling Simple Merge Conflicts

What Are Merge Conflicts?

Conflicts occur when Git can't automatically combine changes from different branches.

When You See a Conflict

```
def train_model(data, params):
<<<<<<< HEAD
    # Your current version
    model = RandomForestClassifier(n_estimators=100)
=======
    # Incoming version
    model = RandomForestClassifier(n_estimators=200, max_depth=10)
>>>>>>> feature/model-improvements
    return model
```

Resolution Steps

1. **Edit the file** to choose what to keep
2. **Remove conflict markers** (`<<<<<<<` , `=======` , `>>>>>>>`)
3. **Stage the resolved file**: `git add filename.py`
4. **Commit the merge**: `git commit -m "Resolve merge conflict in model training"`

---

6. Basic Pull Requests

What is a Pull Request?

A formal way to propose changes to a repository and get them reviewed by teammates.

Simple PR Workflow

```
graph TD
    A[Create Feature Branch] --> B[Make Changes]
    B --> C[Push to Remote]
    C --> D[Open Pull Request]
    D --> E[Team Review]
    E --> F{Approved?}
    F -->|Yes| G[Merge to Main]
    F -->|No| H[Request Changes]
    H --> B
    G --> I[Delete Feature Branch]
```

1. **Create feature branch**: `git checkout -b feature/new-feature`
2. **Make changes and commit**: `git commit -m "Add new feature"`
3. **Push branch**: `git push origin feature/new-feature`
4. **Create PR** on GitHub/GitLab
5. **Get review and merge**
6. **Delete feature branch**

Basic PR Description Template

```
## What this PR does
Brief description of changes

## Changes made
```

```
- Added feature X
- Fixed bug Y
- Updated documentation

## Testing
- [ ] Code runs without errors
- [ ] Tests pass locally
```

---

7. Essential Git Commands

Daily Workflow

```
# Start work - sync with team
git checkout main
git pull origin main

# Create feature branch
git checkout -b feature/my-work

# Check what's changed
git status
git diff

# Stage and commit
git add .
git commit -m "Descriptive commit message"

# Push work
git push origin feature/my-work
```

Checking History

```
# See commit history
git log --oneline

# See what changed in last commit
git show

# See differences
git diff  # Working directory vs staging
git diff --cached  # Staging vs last commit
```

---

8. Understanding Git References

Commit History Navigation

Git uses special references to navigate through your project's commit history. Here's how it looks:

```
a7b8c9d <- HEAD (current commit)        "Fix model accuracy bug"
5e6f7a8 <- HEAD~1 (one commit back)     "Add feature engineering"
2c3d4e5 <- HEAD~2 (two commits back)    "Initial data preprocessing"
9f0a1b2 <- HEAD~3 (three commits back)  "Project setup"
```

HEAD and Commit Navigation

```
# HEAD points to the current commit
git show HEAD

# HEAD^ or HEAD~1 points to the parent commit (one commit back)
git show HEAD^
git show HEAD~1

# HEAD~2 points to two commits back
git show HEAD~2

# HEAD~3 points to three commits back
git show HEAD~3
```

Key Concept: Think of `HEAD` as "where you are now" in your project's timeline. `HEAD~1` is "yesterday" and `HEAD~2` is "two days ago".

SHA-1 Hashes - Git's Internal Names

Every commit, file, and object in Git has a unique SHA-1 hash (40-character identifier):

```
# Full commit hash (40 characters)
git log
# Example: commit 3a5b8c9d2e1f4a7b6c8d9e0f1a2b3c4d5e6f7a8b

# Short hash (first 7 characters, usually sufficient)
git log --oneline
# Example: 3a5b8c9 Add data preprocessing pipeline

# Use hashes to reference specific commits
git show 3a5b8c9
git checkout 3a5b8c9d2e1f4a7b6c8d9e0f1a2b3c4d5e6f7a8b
```

Key Concept: SHA-1 hashes are like fingerprints - each commit has a unique identifier that never changes.

Git Reset with HEAD

Git reset allows you to move HEAD and optionally modify your working directory and staging area:

```
# Soft reset - move HEAD, keep staging area and working directory
git reset --soft HEAD~1
# Effect: Undo last commit, but keep changes staged for re-commit

# Mixed reset (default) - move HEAD, reset staging area, keep working directory
```

```
git reset HEAD~1
git reset --mixed HEAD~1
# Effect: Undo last commit and unstage changes, but keep files modified

# Hard reset - move HEAD, reset staging area AND working directory
git reset --hard HEAD~1
# Effect: Completely undo last commit, lose all changes (DANGEROUS!)
```

Visual Example:

```
Before reset:
a7b8c9d <- HEAD        "Fix model accuracy bug"
5e6f7a8 <- HEAD~1      "Add feature engineering"
2c3d4e5 <- HEAD~2      "Initial data preprocessing"

After git reset --hard HEAD~1:
5e6f7a8 <- HEAD        "Add feature engineering"
2c3d4e5 <- HEAD~1      "Initial data preprocessing"
# The "Fix model accuracy bug" commit is gone!
```

Use Cases:

- **Soft reset**: Fix commit message or combine commits
- **Mixed reset**: Unstage files accidentally added
- **Hard reset**: Completely undo experimental changes (be careful!)

Tags - Human-Friendly Names

Tags provide memorable names for important commits:

```
# Create a tag for current commit
git tag v1.0

# Create an annotated tag with message
git tag -a v1.0 -m "First stable model"

# List all tags
git tag

# Show tag information
git show v1.0

# Checkout a specific tag
git checkout v1.0

# Push tags to remote
git push origin v1.0
git push origin --tags  # Push all tags
```

Data Science Tag Examples:

- **Model versions**: `model-v1.0-baseline`, `model-v2.0-improved`

- **Data releases**: `data-v1.0`, `experiment-results-2024`
- **Milestones**: `paper-submission`, `production-ready`

```
# Tag your best model
git tag -a model-v1.2 -m "Random Forest with 87% accuracy

- Hyperparameter tuned
- Feature selection applied
- Cross-validation tested
- Ready for production"

# Later, easily return to this exact version
git checkout model-v1.2
```

---

9. Basic .gitignore

Create a `.gitignore` file to exclude files from Git:

```
# Python
*.pyc
__pycache__/
.ipynb_checkpoints/

# Data files
data/
*.csv
*.parquet

# Models
*.pkl
*.model

# Environment
.env
venv/
```

---

10. Common Issues and Quick Fixes

"I committed to wrong branch"

```
# Create branch from current position
git branch feature/my-work

# Reset main to clean state
git checkout main
git reset --hard origin/main
```

```
# Switch to feature branch
git checkout feature/my-work
```

"I need to undo last commit"

```
# If not pushed yet - amend
git commit --amend

# If already pushed - revert
git revert HEAD
```

"My branch is behind main"

```
git checkout main
git pull origin main
git checkout feature/my-branch
git merge main
```

---

11. Quick Reference

Essential Commands

```
git status                    # Check current state
git add filename             # Stage specific file
git add .                    # Stage all changes
git commit -m "message"      # Commit with message
git push origin branch-name  # Push to remote
git pull origin main         # Get latest changes

git checkout -b new-branch   # Create and switch to branch
git checkout branch-name     # Switch to existing branch
git merge branch-name        # Merge branch into current
git branch -d branch-name    # Delete branch
```

Workflow Summary

1. **Start**: `git checkout main && git pull origin main`
2. **Branch**: `git checkout -b feature/my-work`
3. **Work**: Edit files, `git add .`, `git commit -m "message"`
4. **Share**: `git push origin feature/my-work`
5. **Merge**: Create Pull Request, get review, merge
6. **Clean**: `git checkout main && git branch -d feature/my-work`

Key Takeaways

1. **Always work on feature branches** - never directly on main
2. **Commit frequently** with descriptive messages

3. **Pull before starting new work** to stay synchronized
4. **Use Pull Requests** for all changes to get team review
5. **Clean up branches** after merging to keep repository tidy

Remember: Git becomes intuitive with practice. Start with these basics and gradually learn more advanced features as your team's needs grow.