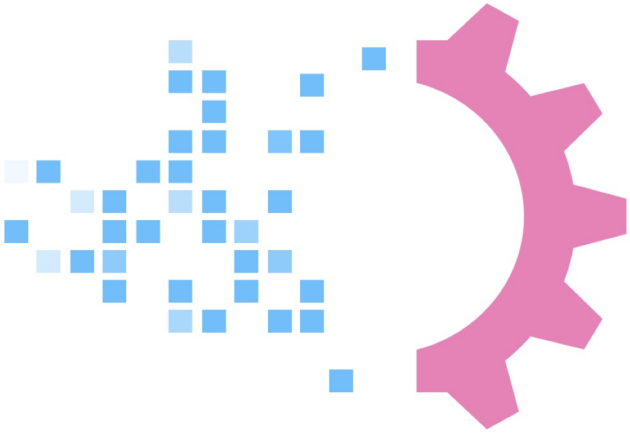
PRODUCTION READY DATA SCIENCE



FROM PROTOTYPING TO PRODUCTION WITH PYTHON

KHUYEN TRAN

CodeCut

Table of contents

Preface

Motivation

Have you ever encountered these situations in your data science projects?

- Your Jupyter Notebook starts simple but becomes a mess as the project grows
- Debugging takes forever because code is scattered and poorly organized
- Package installations break your environment and waste hours troubleshooting
- Code is difficult to adapt to new datasets or requirements
- Code fails to run consistently across different environments
- Changes are hard to track and rollback to previous working versions
- Previously written code is challenging to reuse and extend
- Critical bugs surface late in development
- Adding new features feels risky due to potential regressions

These challenges arise from the gap between exploratory data analysis and production-grade software engineering practices. This book aims to bridge this gap.

The book covers a wide range of essential topics for building production-ready data science applications. Here's an overview of what you'll learn:

1. **Version Control for Code:** Explore version control systems like Git and learn how to apply version control practices to your code, enabling you to track changes, collaborate with others, and manage your codebase effectively.
2. **Dependency Management:** Learn how to handle Python package dependencies using tools like pip or poetry, ensuring consistent and reproducible environments for your projects.
3. **Python Modules and Packages:** Master the creation, organization, and use of Python modules and packages to structure your code efficiently and promote reusability.
4. **Python Variables, Functions, and Classes:** Learn techniques for writing clean and modular code using variables, functions, and classes, enabling better code organization and reusability.
5. **Unit Testing:** Learn how to write effective unit tests using frameworks like

pytest, enabling you to catch bugs early, improve code quality, and facilitate future code changes.
6. **Project Configuration:** Learn how to separate configuration parameters from code logic, allowing for easier customization and deployment across different environments.
7. **Logging and Exception Handling:** Learn how to generate informative log messages that aid debugging, troubleshooting, and monitoring application behavior.
8. **Data Validation:** Discover techniques for validating data types, ranges, formats, and consistency, enabling you to build more reliable and robust data science pipelines.
9. **Version Control for Data:** Learn strategies and tools for versioning your data, ensuring reproducibility and traceability in your data science projects.
10. **Packaging Projects:** Discover how to structure your project for distribution, create setup files, and publish your package to PyPI, making it easy for others to install and use your code.
11. **Building a CI Pipeline:** Learn how to set up a Continuous Integration (CI) to automate code testing and documentation generation, ensuring code quality and facilitating collaborative development.
12. **Jupyter Notebook Best Practices:** Master techniques for creating well-structured, reproducible, and shareable Jupyter notebooks, including cell organization, markdown usage, and version control integration.

Audience

The primary audience for this book includes:

1. **Data Scientists**: Professionals who are skilled in data analysis, machine learning, and statistical modeling, but may lack experience in software engineering practices necessary for production environments.
2. **Data Analysts**: Those who work with data and create analyses but want to improve the scalability and maintainability of their projects.
3. **Machine Learning Engineers**: Professionals who are looking to bridge the gap between creating models and deploying them in production environments.
4. **Data Science Students**: Advanced students or recent graduates who want to learn practical skills for transitioning from academic projects to industry-standard practices.
5. **Research Scientists**: Those in academia or research institutions who want to make their work more reproducible and easier to collaborate on.
6. **Data Science Team Leads**: Professionals responsible for improving their team's workflow and code quality.

Prerequisites

- Familiarity with fundamental Python concepts, syntax, and data structures.
- A foundational understanding of basic data science concepts, such as data processing and model training.
- Basic knowledge of using the command-line interface for tasks like navigating directories and running scripts.
- Basic familiarity with popular data science tools like pandas, NumPy, and matplotlib would be beneficial but not mandatory.

What Makes This Book Different

1. **Simplified Language:** The book materials are presented in a manner that is easy to understand, making complex concepts more accessible to learners.
2. **Visual Support:** Clear and visually appealing graphs and examples accompany each concept and topic, enhancing understanding and providing visual aids for better retention.
3. **Practical Examples:** The examples provided are directly related to data science projects, offering practical applications for the concepts discussed.

About the Author

Khuyen Tran transforms how data scientists learn and work. She has written over 180 articles as a top writer on Towards Data Science, helping data professionals bridge the gap between prototyping and production.

As founder of CodeCut, she publishes daily Python tips in her newsletter that reach over 10,000 views per month and has built a community of 110,000 LinkedIn followers.

Previously an MLOps Engineer and Senior Data Engineer at Accenture, she built enterprise data solutions for clients worldwide.

Copyright

First Edition

Published: January 2025

Published by: CodeCut Technologies LLC

Author: Khuyen Tran

Contact: khuyentran@codecut.ai or visit codecut.ai

Chapter 1

Version Control

1.1 What Is Version Control?

Version control is a system that tracks changes to files and enables software developers to collaborate in a safe, organized, and effective way. Version control allows teams to manage their codebase efficiently, revert changes when needed, and safely experiment with code changes.

1.2 Why Is Version Control Essential?

Version control is especially important in a data science project for several key reasons.

1.2.1 Track Changes and Revert Easily

Version control provides safety and efficiency by tracking every code change, allowing quick recovery when problems occur, and maintaining a complete project history.

Consider developing a machine learning model for customer churn prediction without version control. After making significant changes to your `model.py` file, testing shows degraded performance. Without an accurate record of changes, you spend hours manually trying to undo modifications, risking new errors in the process.

Version control solves this by letting you commit changes regularly during development. When testing reveals performance drops, you can review commit history, identify the problematic change, and revert to the previous working state, as shown in Figure 1.1.

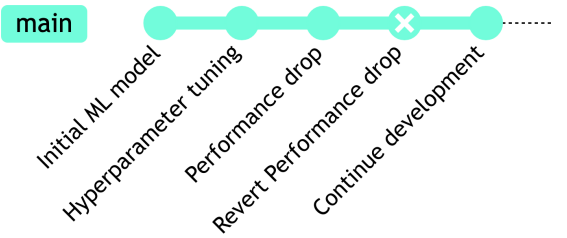


Figure 1.1: Version control workflow for machine learning model development

1.2.2 Collaborate Effectively

Version control transforms team collaboration by eliminating file conflicts, tracking contributor changes, and enabling organized project coordination.

Consider a data analysis project with multiple team members where each person saves work on their local machine or shared drive. Combining everyone's work creates conflicting file versions and overwritten changes. You spend hours manually merging different code versions, trying to reconcile discrepancies and ensure nothing is lost.

Version control provides a shared repository where each team member works on their own branch without affecting the main codebase. Changes are tracked with contributor information and timestamps, enabling safe merging back into the main branch, as illustrated in Figure 1.2.

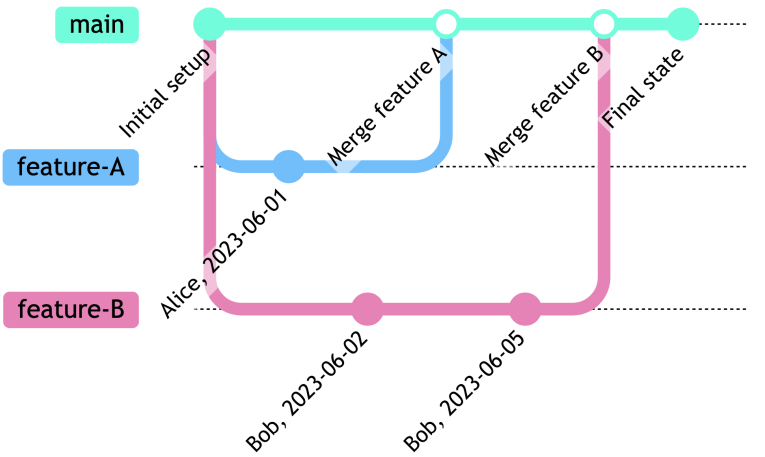


Figure 1.2: Branching and merging workflow in version control

1.2.3 Reproduce Results Reliably

Version control delivers reliable research reproduction by tracking exact code versions and removing guesswork about which files generated specific results.

Consider this scenario: you publish a machine learning model, then need to reproduce results six months later. Without version control, you find multiple script copies with slight variations but can't identify which version created the published results.

Version control eliminates this uncertainty by letting you tag the exact code version used for publication. When you need to reproduce results, you simply checkout the tagged version to recreate the analysis, as shown in Figure 1.3.

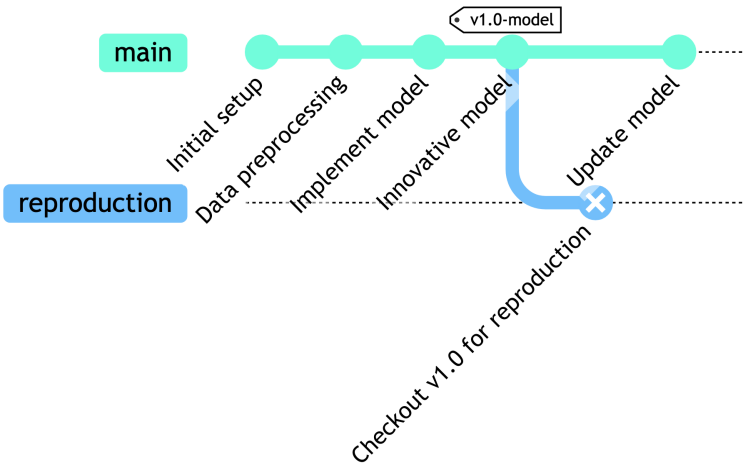


Figure 1.3: Reproducing analysis using a tagged version of code

1.2.4 Experiment Safely

Version control eliminates the fear of experimentation by providing a safety net for code changes. Instead of risking production systems with direct modifications, you can test ideas in isolation, maintain system stability, and recover quickly from failed experiments.

Consider this scenario: you're working on a production data workflow that processes customer data daily and want to test an optimization. Without version control, you make changes directly to production code. Your experiment fails, breaking the system and disrupting daily data flow while you struggle to revert changes under pressure.

With version control, you create a new branch called `feature/new-processing` and freely experiment with your new ideas where changes won't affect the main production code. After thoroughly testing your experiment, you create a pull request to

merge your changes into the `main` branch.

If your changes don't work, you simply discard the experimental branch without affecting the production code, as shown in Figure 1.4, allowing you to innovate without fear of breaking the existing system.

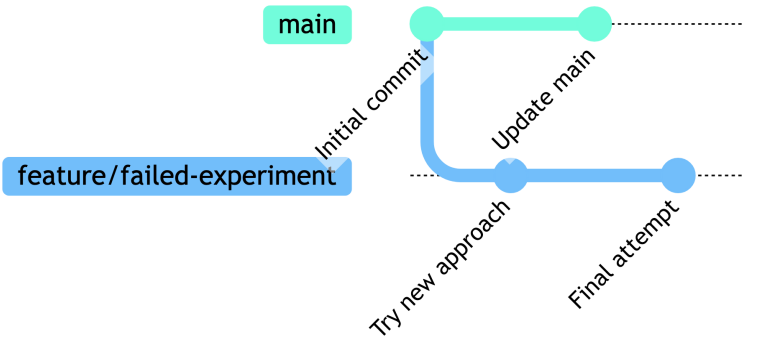


Figure 1.4: Discarding failed experimental branch

1.2.5 Backup Your Project Securely

Version control protects your work by creating automatic backups and preserving your project history. You can recover from computer crashes, accidental deletions, and other disasters without losing progress.

Imagine working on a data science project for weeks, making steady progress on your code and files. Your computer crashes, and you realize you don't have a backup of your project. You've lost all your hard work and must start from scratch. This becomes a frustrating and time-consuming process that sets your project back significantly.

Version control solves this by letting you create a repository for your project and commit changes regularly. The repository serves as both project history and backup, keeping your code safe. If your computer crashes or you accidentally delete a file, you simply restore the project from the repository, as illustrated in Figure 1.5.

1.3 Use Git for Version Control

To implement effective version control and reap its benefits, developers need a robust tool. This is where Git comes into play. Git is a free open-source version control tool that's ubiquitous and trusted by developers worldwide.

1.3.1 Key Git Concepts

Before diving into Git usage, let's understand some key terminology:

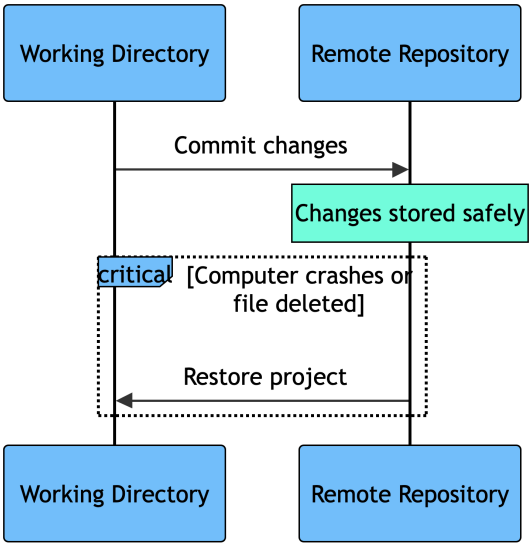


Figure 1.5: Restoring a project from a remote repository

- **Working Directory**: This is the directory on your computer where you're actively working on your project files.
- **Local Repository**: This is a hidden `.git` folder in your working directory that contains the complete history of your project. When you commit changes, they are stored here.
- **Remote Repository**: This is a version of your project hosted on a server (like GitHub or GitLab). The Remote Repository allows you to back up your code and collaborate with others. You can push changes to and pull updates from your repository.

Figure 1.6 illustrates the components of Git version control.

Let's explore how we can effectively use Git in different scenarios.

1.3.2 Scenario 1: Starting a New Project

When beginning a new data science project, establishing version control from the start creates a solid foundation for development. This scenario covers the complete workflow from initializing Git in your project directory to connecting with a remote repository for backup and collaboration.

1.3.2.1 Overview

This workflow involves three main phases:

- Creating the local repository

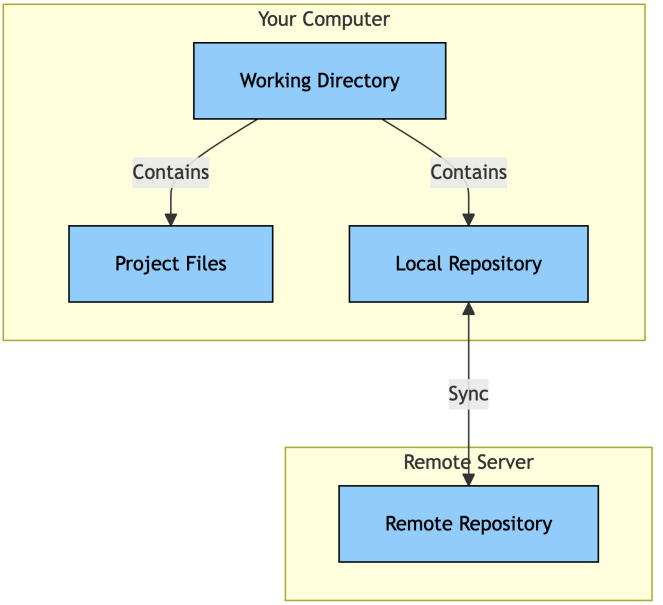


Figure 1.6: Components of Git Version Control

- Making your first commit
- Connecting to a remote repository for backup

1.3.2.2 Step-by-Step Process

Phase 1: Initialize Local Repository

1. Initialize a new Git repository in your working directory:

```
git init
```

Phase 2: Create First Commit

2. Stage the changes or new files in your Git repository:

```
# Add all changes and new files
git add .
```

3. Review the list of changes to be committed:

```
git status
```

```
Changes to be committed:

      new file:   .gitignore
      new file:   .pre-commit-config.yaml
      ...
```


4. Save the staged changes permanently in your local repository's history along with a commit message:

```
git commit -m 'init commit'
```

Phase 3: Connect to Remote Repository

5. Create a repository on GitHub/GitLab and add the remote connection. If you're using GitHub as the remote repository, create a new repository on GitHub and copy its URL. Then, add the URL to your local Git repository with the name `origin`:

```
git remote add origin <repository URL>
```

6. Push your initial commit to establish the remote backup:

```
# Push to the main branch on the origin repository
git push origin main
```

Figure 1.7 illustrates this workflow.

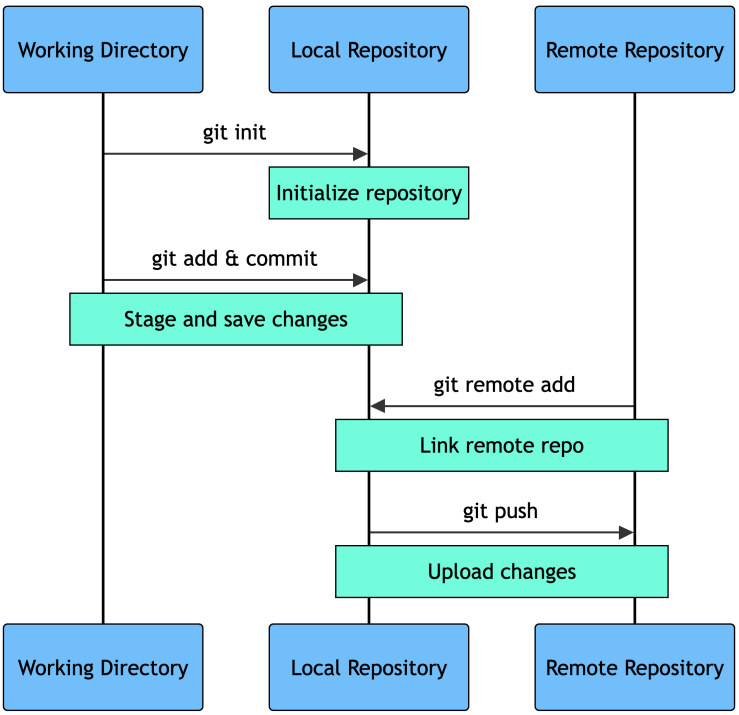


Figure 1.7: Initializing a Git repository and uploading the project to a remote repository

1.3.3 Scenario 2: Contributing to an Existing Project

When you want to contribute to an existing data science project, whether it's an open-source library or your team's codebase, you need to safely integrate your changes without disrupting the main project. This scenario covers the complete workflow from forking and cloning to submitting your contributions through pull requests.

1.3.3.1 Overview

This workflow involves four main phases:

- Getting access to the project code
- Setting up your local development environment
- Making and testing your changes
- Submitting your contributions for review

1.3.3.2 Step-by-Step Process

Phase 1: Get Access to the Project Code

1. Fork the repository on GitHub if you don't have write access to the main repository.
2. Use `git clone` to create a local copy of the remote repository on your machine.

```
git clone https://github.com/username/project-name.git
```

Phase 2: Set Up Your Development Environment

3. Navigate to the project directory:

```
cd project-name
```

4. Create and switch to a new branch to safely develop your changes without affecting the main codebase:

```
git checkout -b <branch-name>
```

Phase 3: Implement Your Changes

5. Make your code modifications in the new branch.
6. Stage, commit, and push your changes:

```
git add .
git commit -m "Descriptive message about your changes"
git push origin <branch-name>
```

Phase 4: Submit Your Contribution

7. Create a pull request on GitHub to propose merging your changes. This enables project maintainers to review your contributions before integrating them into the main project.

Figure 1.8 illustrates this process.

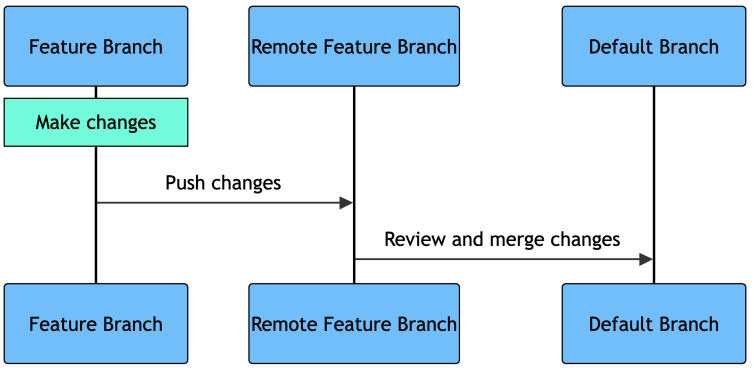


Figure 1.8: Pull request workflow

1.3.4 Scenario 3: Staying Synchronized

When working on a team project or contributing to an active repository, the main branch often receives updates while you're developing your features. This scenario covers how to keep your local work synchronized with remote changes to avoid conflicts and maintain a current codebase.

1.3.4.1 Overview

This workflow involves two main phases:

- Securing your current work
- Integrating remote updates

1.3.4.2 Step-by-Step Process

Phase 1: Secure Your Current Work

1. Ensure your local work is saved by staging and committing your local changes. This prevents losing your progress:

```
git add .
git commit -m 'commit-2'
```

Phase 2: Integrate Remote Updates

2. Pull changes from the remote main branch with `git pull`, which creates a merge commit combining your work with the latest updates:

```
git pull origin main
```

Figure 1.9 illustrates this process.

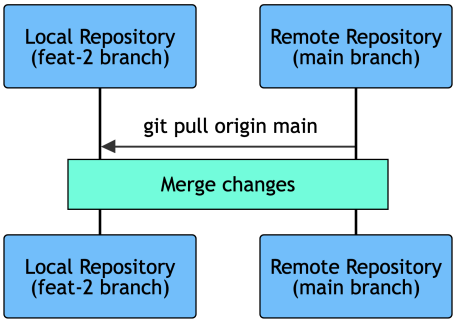


Figure 1.9: Merging remote changes from the main branch into the local feat-2 branch

1.4 Best Practices in Version Control

1.4.1 Error Recovery and History Management

Have you ever pushed a commit and immediately realized it contained a bug? When you need to undo changes in a shared repository, choosing the right recovery method prevents disrupting team workflows and maintains project integrity.

Git provides two main approaches for handling these situations:

- **Safe recovery with `git revert`**: Creates new commits that undo changes, preserving complete history
- **History rewriting with `git reset`**: Moves branch pointer to different commits, effectively rewriting history

1.4.1.1 When You Need to Preserve History

Use `git revert` when:

- Commits have been pushed to shared repositories
- Working in team environments where history preservation is important
- You want to maintain a complete audit trail of changes

To revert a specific commit, first identify the commit hash:

```
git log
```

```
commit 0b9bee172936b45c3007b6bf6fa387ac51bdeb8c
    commit-2
```



```
commit 992601c3fb66bf1a39cec566bb88a832305d705f
    commit-1
```

Then use `git revert` with the commit hash:

```
git revert 992601c3fb66bf1a39cec566bb88a832305d705f
```

Figure 1.10 illustrates the git revert process.

1.4.1.2 When You Need to Remove Commits

Use `git reset` when:

- Commits exist only in your local repository
- You need to completely remove commits from history
- Working on private feature branches before sharing

To reset commits, identify the target commit hash with `git log`, then choose your reset type:

```
# Soft reset: Keep changes staged
git reset --soft <commit-hash>

# Mixed reset: Keep changes unstaged (default)
git reset <commit-hash>

# Hard reset: Discard all changes
git reset --hard <commit-hash>
```

Figure 1.11 illustrates the git reset process.

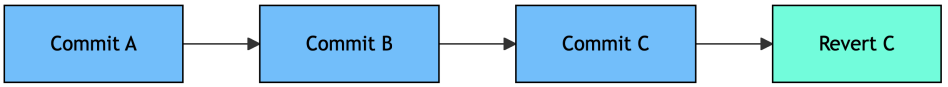


Figure 1.10: Git revert creates new commit to undo changes

Figure 1.11: Git reset removes commits from history

> **Warning**
>
> Warning: Unlike `git revert`, `git reset` rewrites commit history. Never use `git reset` on commits that have been pushed to a shared repository, as this

> can cause problems for other team members.

1.4.2 Managing Uncommitted Work

Have you ever been deep in coding when you suddenly need to pull updates from the remote repository, but your changes aren't ready to commit? Properly managing work-in-progress prevents lost changes and maintains clean development workflows.

Git stash provides a solution for temporarily storing uncommitted changes:

- **Temporary storage**: Save current changes without creating commits
- **Clean workspace**: Switch branches or pull updates safely
- **Easy restoration**: Reapply stashed changes when ready to continue

For example, when you have uncommitted changes but need to pull updates:

```
git status
```

```
On branch feat-2
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
        modified:   file2.txt
```

Use `git stash` to temporarily save your changes:

```
git stash
```

Now your working directory is clean:

```
git status
```

```
On branch feat-2
nothing to commit, working tree clean
```

You can safely pull updates:

```
git pull origin feat-2
```

After pulling, reapply your stashed changes:

```
git stash pop
```

```
On branch feat-2
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
        modified:   file1.txt
```

```
        modified:   file2.txt
Dropped refs/stash@{0} (1234abcd5678efgh)
```

Figure 1.12 illustrates this process.

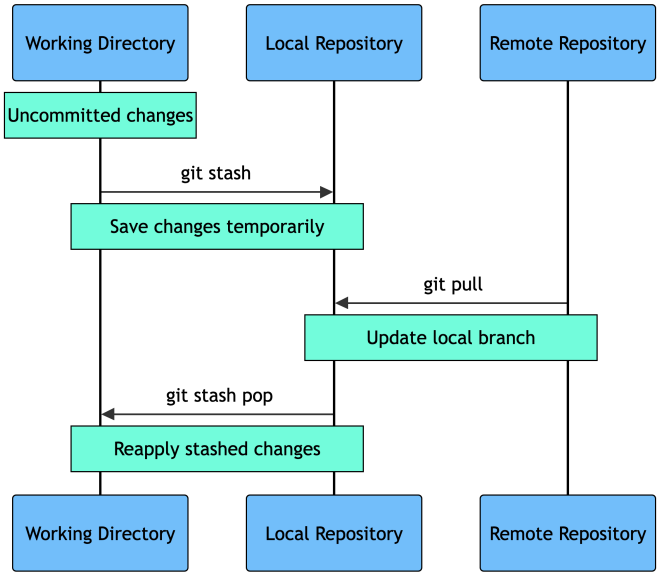


Figure 1.12: Stashing and reapplying changes during a pull operation

1.4.3 Ignore Large and Private Files

Have you ever tried to clone a repository only to wait ages for a massive download to complete? When developers include large datasets or confidential credentials in their Git repository, it creates bloated repositories that are slow, insecure, and difficult to share.

Git's ignore functionality solves this by letting you specify which files to exclude from version control. This helps:

- Keep repositories small and efficient
- Protect sensitive information
- Reduce unnecessary version tracking of large binary files

Create a `.gitignore` file in your project's root directory to specify which files and directories Git should ignore (shown in Example 11.2).

Example 1.1 .gitignore

```
# Ignore large data files
*.csv
*.parquet
*.feather
*.h5

# Ignore model files
*.pkl
*.joblib
*.pt

# Ignore sensitive information
.env

# Ignore Jupyter notebook checkpoints
.ipynb_checkpoints/

# Ignore virtual environment
venv/
env/

# Ignore IDE files
.vscode/
```

1.4.4 Commit Often and Logically

Have you ever struggled to understand what changed in a massive commit? Large commits mixing unrelated changes make it difficult to review, understand, and selectively revert specific modifications.

When commit focuses on a specific aspect of the project, it becomes easier to:

- Track changes and their impact
- Review code modifications
- Revert specific changes if needed
- Understand the project's evolution
- Communicate what each commit does to other team members

Here are some examples of small commits with clear, descriptive messages:

```
# Commit 1: Data preprocessing
git commit -m "Add data cleaning and preprocessing steps"

# Commit 2: Feature engineering
git commit -m "Create new features for customer churn prediction"
```



```
# Commit 3: Model training
git commit -m "Train initial random forest model"

# Commit 4: Model evaluation
git commit -m "Add detailed model evaluation"

# Commit 5: Model improvement
git commit -m "Optimize model hyperparameters"
```

1.4.5 Fetch Before Merge

Have you ever run `git pull` only to find unexpected merge conflicts or mysterious code changes in your working directory? Using `git pull` automatically merges remote changes without review, causing unexpected conflicts and unintentional changes.

Instead, use `git fetch` followed by `git merge` to examine incoming changes before merging. You can review new commits, check for conflicts, and decide when and how to integrate updates into your work.

Here are the steps to fetch and merge remote changes:

1. Fetch the latest changes from the remote repository without modifying your working directory.

   ```
   git fetch origin
   ```

2. Review the changes:

   ```
   # Check differences between current branch and remote main
   git log ..origin/main
   ```

   ```
   e4f5g6h Update data preprocessing script
   d7e8f9a Add new feature extraction function
   ```

3. Once you're satisfied with the changes, merge them:

   ```
   git merge origin/main
   ```

   ```
   Updating a1b2c3d..e4f5g6h
   Fast-forward
   preprocessing.py | 15 ++++++++++++---
   feature_extraction.py | 25 +++++++++++++++++++++++++
   2 files changed, 37 insertions(+), 3 deletions(-)
   ```

Figure 1.13 demonstrates this process.

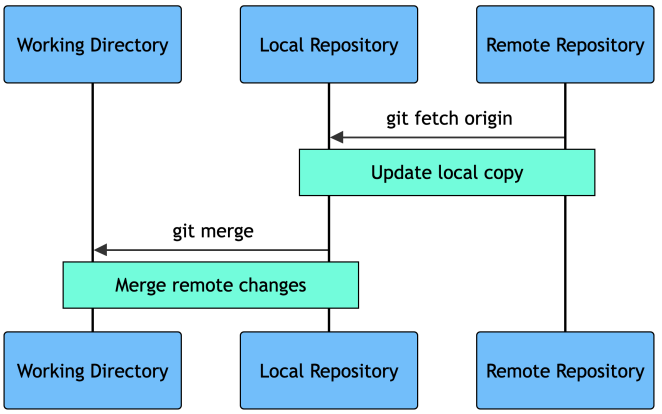


Figure 1.13: Fetching and merging remote changes

1.5 Key Takeaways

Version control is an essential tool for data scientists, enabling efficient collaboration, experimentation, and project management. Here are the key takeaways from this chapter:

1. Core benefits of version control:
    - Track changes and revert to previous versions when needed
    - Collaborate effectively with team members
    - Reproduce results reliably
    - Experiment safely without affecting production code
    - Backup your project safely and securely
2. Git fundamentals:
    - Working Directory: Where you make changes to files
    - Local Repository: Stores complete project history
    - Remote Repository: Hosts code for backup and collaboration
    - Commits: Snapshots of changes with descriptive messages
    - Branches: Isolated environments for feature development
3. Essential Git commands:
    - `git init`: Start a new repository
    - `git add` & `git commit`: Save changes
    - `git push` & `git pull`: Sync with remote repository
    - `git branch` & `git checkout`: Manage different versions
    - `git merge`: Combine changes from different branches
    - `git revert` & `git reset`: Undo changes when needed
4. Best practices:
    - Use `.gitignore` to exclude large files and sensitive data
    - Make small, focused commits with clear messages
    - Fetch before merging to review changes
    - Create feature branches for new development

- Use `git revert` for shared repositories, `git reset` only locally
- Use `git stash` for work-in-progress storage
- Regularly sync with the remote repository

By following these practices and understanding these concepts, you can effectively manage your code, collaborate with others, and maintain a clean, organized project history.