Version Control and Further Graphs

February 22, 2024

1 Applied Data Science 1

1.0.1 Dr. William Cooper

1.1 Version Control

Code is always changing. How can we control this? By 'versioning'. Several software packages already exist which control and track changes between versions. The most used of these are subversion (SVN) and Git. We will focus in this course on just Git, including the website wrapper, GitHub. If you do not already have an account, you should definitely create one as soon as possible. Most companies/industries will request to view your public GitHub profile to get an idea of your 'portfolio'.

1.2 What is Git?

Git is a version control system that allows multiple people to work on a project at the same time. It helps in tracking changes in source code, enabling multiple contributors to work collaboratively and independently. It is installed in the PC labs on campus. Note that with the command line, it is best to use SSH or CLI for secure authentication, over standard HTTPS links. There are also several different GUIs available with slightly different teminology, but the same pattern follows.

1.2.1 1. Creating a Repository (Repo)

- A repository is like a project folder, containing all your project files and their history.
- **GitHub:** Go to GitHub, click on "New repository", name your repository, and then click "Create repository".
- **GUI:** Use tools like GitHub Desktop. Click on "File" > "New Repository" and follow the instructions.
- **Command Line:** `git init`

1.2.2 2. Cloning a Repository

- Cloning creates a local copy of a remote repository on your computer.
- **GitHub:** Click the green "Code" button on the repository page and copy the URL.
- **GUI:** Select "Clone Repository" and paste the URL.

- **Command Line:** `git clone [URL]`

1.2.3 3. Committing Changes

- A commit is a record of what changes you have made since the last commit.
- **GUI:** After making changes, you'll see a list of modified files. Add a good message describing your changes and click "Commit".
- **Command Line:**

```
git status  # to see the list of changed files
git add <whatever files>  # to stage those files for commiting
# additionally as appropriate
git rm <files>  # deleting files from git
git mv <files>  # moving/renaming files
git commit -m "Your commit message"  # to place the commit
# alternatively
git commit -a -m "Your commit message"  # staging all modified files and placing commit
```

1.2.4 4. Pushing Changes

- Pushing means uploading your committed changes to a remote repository.
- **GitHub:** Click the "Add files" button next to the green "Code" button.
- **GUI:** Click the "Push" button to upload your changes.
- **Command Line:**

```
git remote -v  # see the remote sources like origin and upstream
git push origin [branch-name]  # push the staged commit(s)
# you may need, if there are multiple branches that you want to edit
git push --set-upstream <origin/upstream> [branch-name]
# often, just
git push  # needed
```

1.2.5 5. Pulling Changes

- Pulling is the act of downloading changes from the remote repository to your local repository.
- **GUI:** Click the "Pull" button to fetch and merge changes from the remote repository.
- **Command Line:** `git pull origin [branch-name]` or just `git pull`

1.3 Why Use Git?

- **Collaboration:** Multiple people can work on the same project simultaneously.
- **History Tracking:** Every change is recorded, so you can revert to previous versions if needed.
- **Branching and Merging:** Work on new features or fixes in separate branches without affecting the main project, then merge them when ready.

1.4 Remote, Origin, and Upstream

1.4.1 Remote

- A **remote** in Git is a common repository that all team members use to exchange their changes. It's usually hosted on a server like GitHub.
- In simpler terms, it's a version of your project that resides on the internet or network, where you can push your changes, pull others' changes, and keep the local and remote repositories in sync.

1.4.2 Origin

- **Origin** is the default name Git gives to the server repository you cloned from.
- It's like a bookmark for the remote repository. When you run commands like `git push` or `git pull`, Git knows to push/pull to/from the `origin` remote.
- It's not a fixed name; you can change it or have multiple different remotes for the same local repository.

1.4.3 Upstream

- **Upstream** usually refers to the main project that you forked from on platforms like GitHub.
- When you fork a project, you create your own copy. Setting the original project as `upstream` allows you to keep your fork updated with the changes made in the original repository.
- This is especially important in open-source projects where many contributors work on different aspects simultaneously.

1.5 Why Version Code?

You will have seen this on various softwares, including python, e.g. python 2.7, 3.6, 3.7, 3.8… 3.12. There are a few different ideologies with this but a common one is ... On GitHub this is done by drafting a release or 'tag'.

1. **Track Changes:** Version control keeps a history of who changed what and when. This is crucial for understanding how and why your project evolves over time.
2. **Revert Mistakes:** If something goes wrong, you can easily revert to a previous stable version of your code.
3. **Parallel Development:** Different team members can work on separate features at the same time without interfering with each other. Branches in Git facilitate this parallel development.
4. **Review and Accountability:** Version control allows for code reviews and keeping track of who made specific changes, improving quality assurance.
5. **Document History:** The commit history acts as a documentation of the changes made over time. Each commit message provides context about a particular change.

1.5.1 Merging

- **Merging** is the process of combining changes from different branches into a single branch.
- In GitHub or GUI tools, merging can often be done with a simple click. For example, in a Pull Request on GitHub, you can merge the feature branch into the main branch by clicking "Merge pull request".

Merge Conflicts

- **Merge conflicts** occur when Git is unable to automatically resolve differences in code between two commits.
- Conflicts typically happen when two branches have changed the same part of the same file.
- GUI tools often provide a visual representation of conflicts, making it easier to see and resolve them.
- **Command Line:** You'll need to manually edit the files to resolve conflicts, mark them as resolved (`git add [file]`), and then complete the merge with a commit.

1.5.2 Managing Dependencies with `requirements.txt`

- In Python projects, `requirements.txt` is used to list all the dependencies (libraries/packages) your project needs.
- This file is crucial for version control as it ensures everyone working on the project uses the same versions of dependencies, which reduces compatibility issues.
- To create a `requirements.txt` file, run `pip freeze > requirements.txt` in your project's environment.
- To install dependencies from this file, use `pip install -r requirements.txt`.

1.5.3 Pull Requests (PRs)

- PRs are a feature of platforms like GitHub, where you propose your changes and request that someone reviews and pulls your contribution into their branch.
- PRs are essential for code review and collaboration in a team.

1.5.4 Continuous Integration/Continuous Deployment (CI/CD)

- CI/CD are practices in software development designed to improve code quality and streamline the process of integrating new code and deploying it.
- Tools like GitHub Actions can automate testing and deployment processes.

1.5.5 .gitignore File

- The `.gitignore` file tells Git which files or folders to ignore in a project.
- It's important for keeping your repository clean from unnecessary files like build outputs, temporary files, or sensitive information.

2 Exercise 1

Create a GitHub account if you have not already, and a new repository called 'Applied Data Science 1'. - Write something meaningful in the description box. - Keep as **public**. - Add a `README` (you will write important information in there later). - Add a `.gitignore` (use the Python template). - Add a licence, BSD-3 or MIT are the most typical ones for open source code.

Fill out the `README` form, describing what this repo will be about (it will be a store for your work on this module). *Clone* this repository to your local machine and move this notebook into it. Upload the notebook into the GitHub repository using whatever method. ## End Exercise 1

2.1 Further Graphs

We previously looked at some statistical graphs, but let's go even further today:

[1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

[19]:
```
df = pd.read_csv('Data/measurements.csv', index_col=0).sort_values('phase')
# Assuming this should be a sine wave
x = np.linspace(0.0, 2.0*np.pi, 1000)
y = np.sin(x)
# Let's adopt some typical measurement errors
yerr = 0.05

# Guess at what value should be for each measured value
y_guess = np.sin(df['phase'])
# What are the residuals? How do they compare to the measurement error?
df['residual'] = df['value'] - y_guess
df['significance'] = df.residual.abs() / yerr
df.describe()
```

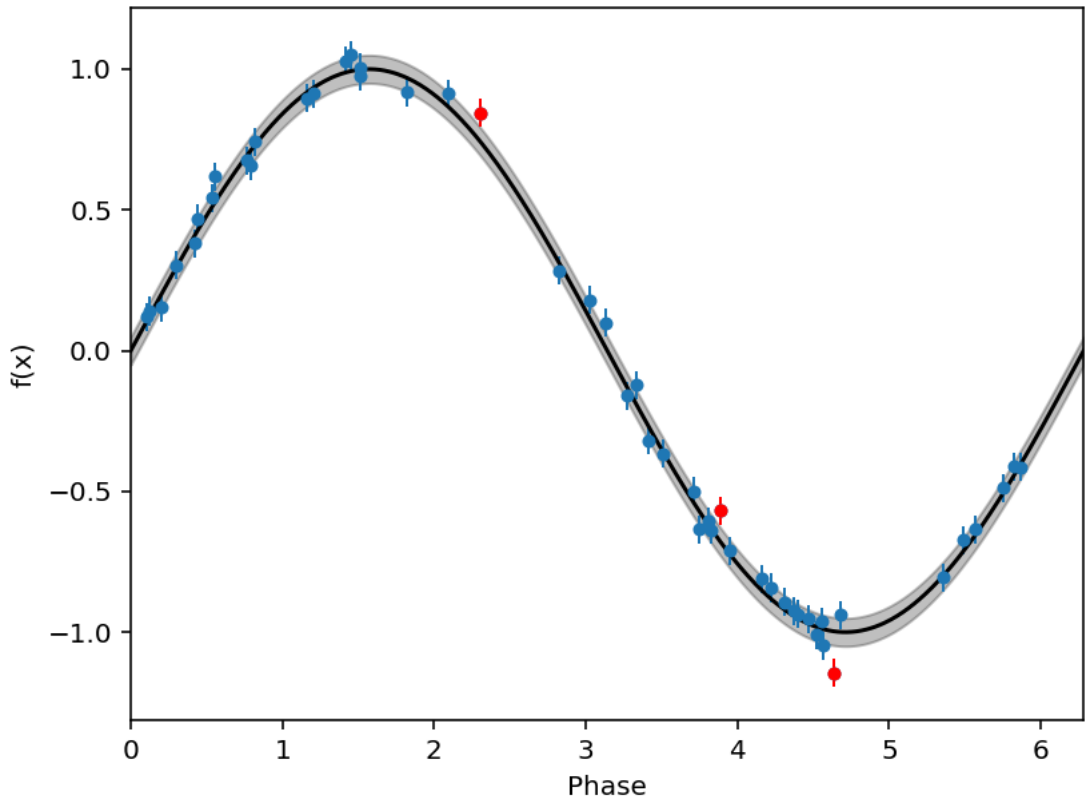
[19]:

	phase	value	residual	significance
count	50.000000	50.000000	50.000000	50.000000
mean	2.954422	-0.092052	0.011121	0.777487
std	1.809022	0.716290	0.048243	0.603825
min	0.101132	-1.145696	-0.148647	0.058257
25%	1.256559	-0.701367	-0.021169	0.349375
50%	3.372297	-0.239826	0.015617	0.575275
75%	4.394863	0.599266	0.041690	1.012382
max	5.867323	1.051524	0.111634	2.972947

[38]:
```
def plot_sine_measurements():
    """
    Plotting the measurements and residuals
```

```
    """
    plt.figure(dpi=144)

    # plot errorbar
    plt.errorbar(df['phase'], df['value'], yerr=yerr, lw=0, elinewidth=1,␣
↪marker='o', ms=4)

    # plot what values should be
    plt.plot(x, y, 'k-')

    # plot error margin of 'model' (typically confidence of predictions)
    plt.fill_between(x, y - yerr, y + yerr, alpha=0.25, color='black')

    # highlight significant outliers
    df_sig = df[df.significance > 2].copy()
    plt.errorbar(df_sig['phase'], df_sig['value'], yerr=yerr, lw=0,␣
↪elinewidth=1, marker='o', ms=4, color='red')

    plt.xlim(x.min(), x.max())
    plt.xlabel('Phase')
    plt.ylabel('f(x)')
    plt.show()
    return
```

[39]:
```
plot_sine_measurements()
```



3 Exercise 2

- Create an array of 200 x values from -10 to 10.
- Define a quadratic function $y = ax^2 + bx + c$. For example, use $a = 1$, $b = 2$, $c = 3$.
- Add random noise to the y values to simulate measurement errors, you can use $\mu = 1$ and $\sigma = 5$.
- Plot these values as a scatter plot with errorbars, with the an expected error of 5 as a confidence band around the true function.

[ ]:
```
def plot_random_quadratic():
    """
    Plots the quadratic with measurement noise
    """
    return
```

[ ]:
```
plot_random_quadratic()
```

3.1 End Exercise 2

To finish, let's think about combining distributions and correlations with pair plots/ corner plots.

[91]:
```
x = np.random.choice(np.linspace(-100, 100, 400), 400)
x2 = np.random.uniform(50, -150, 400)
x3 = np.random.normal(25, 100, 400)
y1 = x * np.random.normal(2, 0.2, x.size)
y2 = x * np.random.normal(2, 0.1, x.size) + x3
y3 = x2 / np.random.normal(2, 0.4, x2.size) - np.random.normal(0, 5, x2.size)
df = pd.DataFrame(data=dict(x=x, x2=x2, x3=x3, y1=y1, y2=y2, y3=y3))
df.corr()
```

[91]:

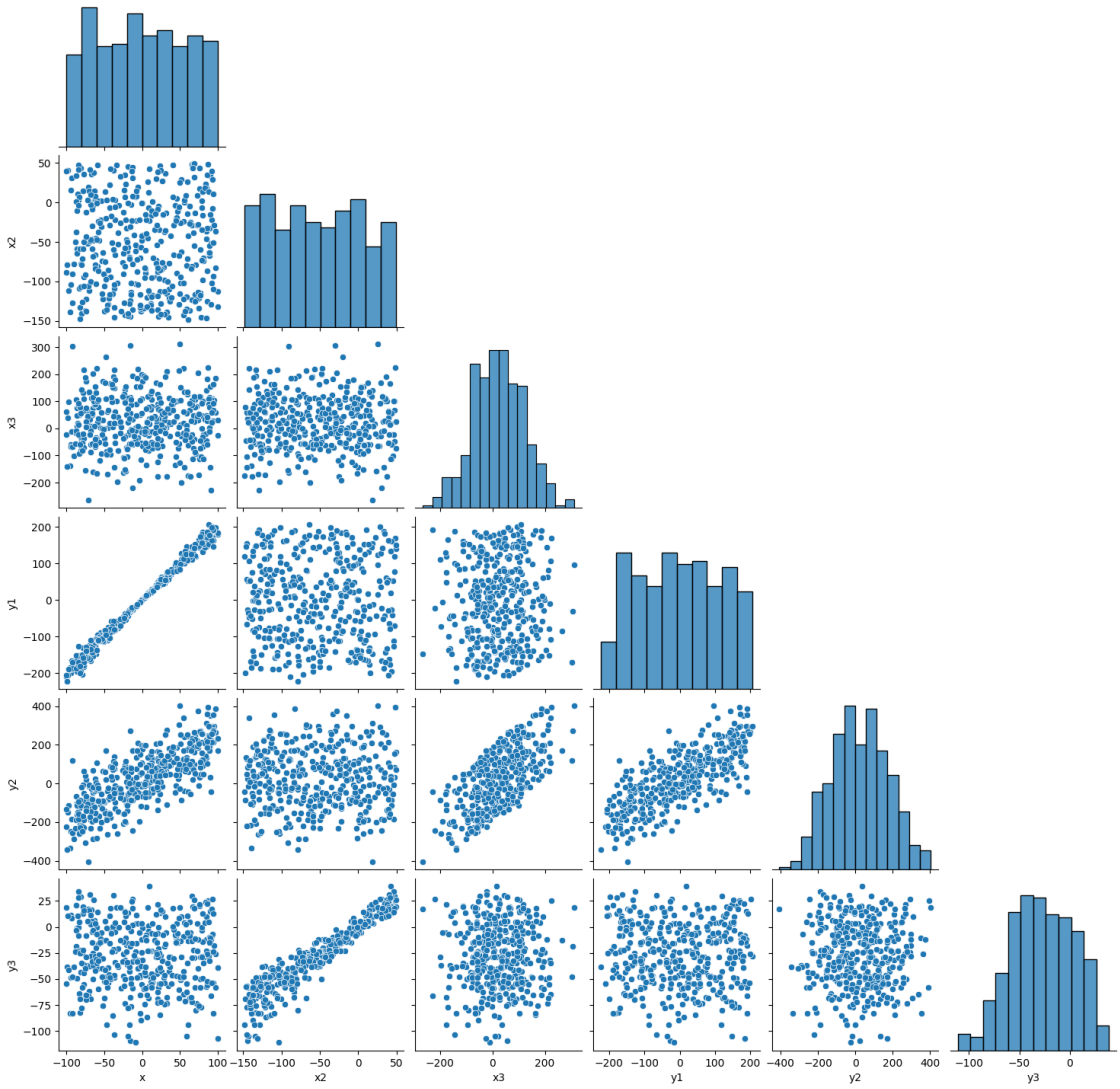
	x	x2	x3	y1	y2	y3
x	1.000000	-0.027502	0.038220	0.995521	0.770264	-0.019238
x2	-0.027502	1.000000	0.008197	-0.032958	-0.015976	0.943006
x3	0.038220	0.008197	1.000000	0.037209	0.665765	0.002158
y1	0.995521	-0.032958	0.037209	1.000000	0.766691	-0.024674
y2	0.770264	-0.015976	0.665765	0.766691	1.000000	-0.013825
y3	-0.019238	0.943006	0.002158	-0.024674	-0.013825	1.000000

[99]:
```
def plot_pairplot(data):
    """
    Create a pairplot of some random correlated data
    """
    sns.pairplot(data, corner=True)
    plt.show()
    return
```

[84]:
```
#  Note that most installations do not have corner installed. Uncomment and run␣
 ↪the command below (but only once)
#  !pip install corner
from corner import corner
```

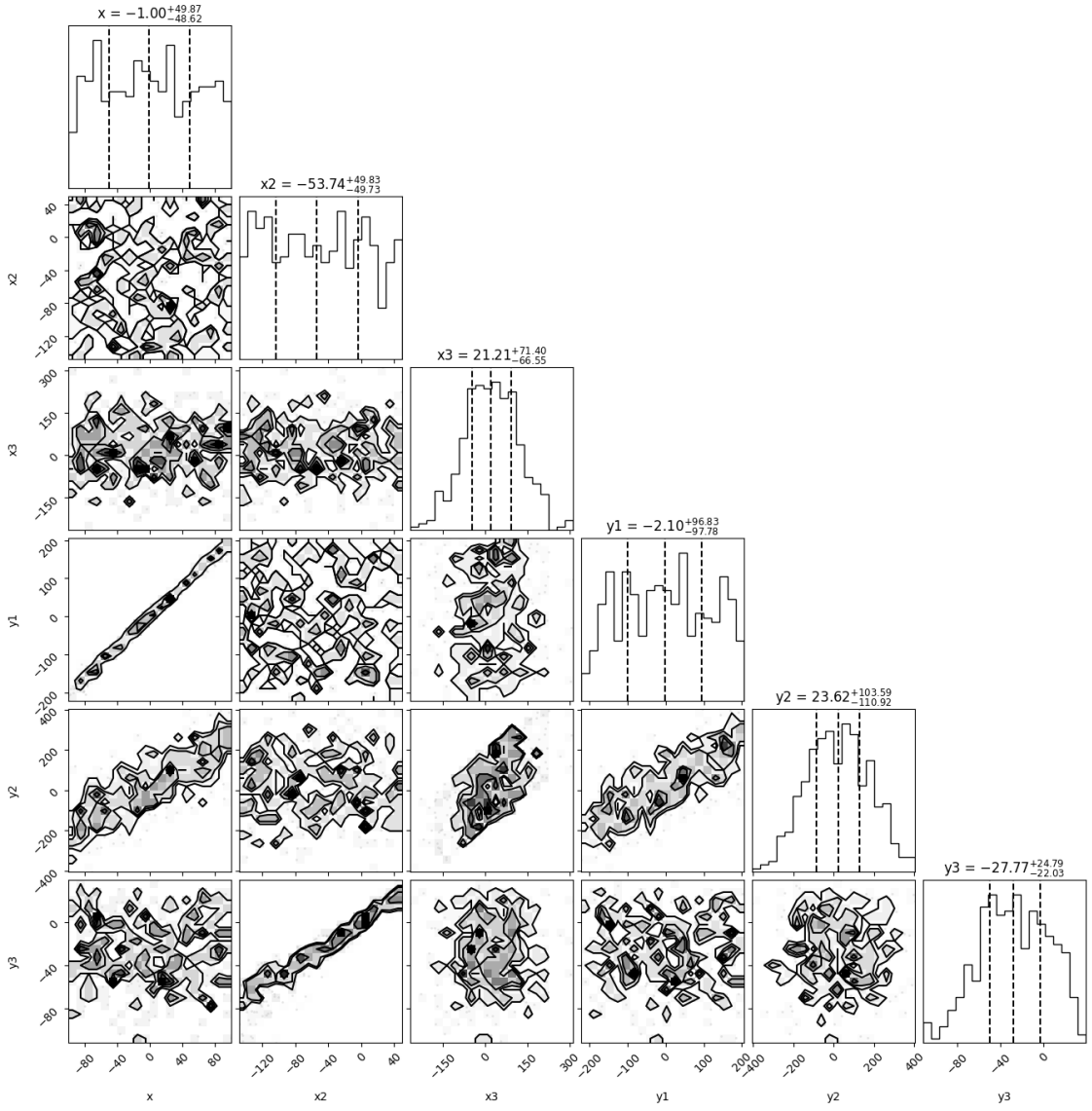
[100]:
```
def plot_cornerplot(data):
    """
    Create a corner plot
    """
    fig = corner(data, show_titles=True, quantiles=[0.25, 0.5, 0.75], labels=df.
 ↪columns)
    plt.show()
    return
```

[101]:
```
plot_pairplot(df)
```



[102]:
```
plot_cornerplot(df)
```

```
WARNING:root:Too few points to create valid contours
WARNING:root:Too few points to create valid contours
```



4 Exercise 3

Use the two samples below and plot them seperately as a pair plot and corner plot. When done, save the notebook and upload it into your new GitHub repository

[ ]:
```
uncorrelated_data = np.random.normal(size=(100, 3))
df_uncorrelated = pd.DataFrame(uncorrelated_data, columns=['X', 'Y', 'Z'])

mean = np.random.uniform(size=3)
random_matrix = np.random.rand(3, 3)
cov = np.dot(random_matrix, random_matrix.T)
```

```
correlated_data = np.random.multivariate_normal(mean, cov, size=100)
df_correlated = pd.DataFrame(correlated_data, columns=['X', 'Y', 'Z'])
```

[ ]:

4.1 End Exercise 3