Data Science Project Architecture

Putting everything together: math, code, data, scientific approach

Yordan Darakchiev

Technical Trainer

iordan93@gmail.com

Table of Contents

- sli.do: [#ds-projects]
- High-quality code and software engineering best practices
  - Code conventions
- Data science project structure
- Improving code
  - Debugging, unit tests, performance tests
- Reproducible research
  - Tools, methods, ideas

High-Quality Code

Best practices, guides, patterns

Code Conventions

- Scientists usually don't care too much about code
- Leads to several things
  - Scientists' code is sometimes hard to understand and maintain
  - Developers can have a hard time debugging, and / or communicating ideas
- Why not take the best of both worlds?
- Python guidelines ("The Zen of Python")

```python
import this
```

- [Google Python Style Guide]()
- It’s good to have code conventions
  - Many people can write code as one
  - {Team / company; language; personal} conventions

General Ideas

- "It's not going to production anyway"
  - Often, this is your production code
- "Why did I write this?"
  - Leave comments, and make your code self-documenting
    - Unit tests can also serve as documentation
    - Other assets (e.g. pdf documents, issues, project requirements, etc.) can also help on a higher level
- Use descriptive names
  - Add meaningful context
  - Avoid misleading names, comments, etc.
- Refactor the code when needed
  - Technical debt
- Separate the code into smaller, single-purpose chunks

Naming Stuff

- `lower_with_under` – variables, functions, files, folders
- `UPPER_WITH_UNDER` – global constants
- `PascalCase` – class names, folders
- `camelCase` – **only** to conform to existing conventions
- Notes
  - `_leading_underscore` – marks a private variable
    - Not truly private, only a signal to developers not to mess with it
    - `__double_leading_underscore` – "mangles" variable names
  - `__double_underscores__` – special variables or methods
    - `__name__`, `__doc__`, `__init__`, `__str__`, `__repr__`, `__len__`, etc.

```
arr = np.array([1, 2, 3])
arr.__str__() # '[1 2 3]'
arr.__repr__() # 'array([1, 2, 3])'
arr.__len__() # 3
```

Readability

- Use `imports` for modules and packages
- Avoid global variables
  - Pollute the global scope
  - Can create subtle dependencies in the code
  - Try using function parameters (and / or classes)
- List comprehensions, lambdas, conditional expressions
  - Okay for simple, one-line cases

```
print([x + 3 for x in range(3)])
sum_two_nums = lambda x, y: x + y
print("even" if a % 2 == 0 else "odd")
```

- Lexical scoping (closures) – use **very** carefully

```
def summator(a): # Usage: summator(4)(5)
    def inner_summator(b):
        return a + b
return inner_summator
```

Readability (2)

- Whitespace
  - **DO NOT** mix tabs and spaces!
    - Prefer spaces (text editors replace 1 tab with 4 spaces by default)
    - This **can create** a lot of pain and **sinister bugs**
  - 1-2 blank lines between variables, functions and methods
  - Use typography rules (e.g. 1 space after comma)
- Comments
  - Avoid inline comments

```
x = x + 1 # Increment x by 1
```

  - Docstrings – a way of documenting the code, unique to Python
    - More info here
  - TODO comments: temporary code, short-term solution, or good enough but not perfect

Object-Oriented Programming

- Python has OOP
- For most of our purposes, it's not needed
  - We have used a lot of objects but we didn't really need to create classes
- We generally prefer a combination of procedural and functional style
- If you're comfortable, feel free to use classes
  - All principles from other OOP languages apply
  - Once again, the goal is to create readable code, which is easier to maintain

Project Structure

How not to get lost

Notebook Structure

- Similar to scientific papers
- Imports – usually the first cell contains all imports
- Title, author(s)
- Abstract – not mandatory, but really good to have
- Data manipulation process
  - Divided into sections and subsections
  - Most commonly: getting data, transformations, visualization, modelling, etc.
- Conclusion(s)
- Tips
  - Make sections self-contained, reduce dependencies
  - Create functions when possible
    - To avoid creating too many global variables

File Structure

- Usually, projects have one notebook
  - You may include many notebooks if you wish
  - You can also import code from notebooks
- Very long code can be separated in `.py` files
  - Not greatly recommended, but sometimes helps
    - E.g. if the file contains a lot of utility functions
- Using: simply import the files
  - Using the file names
  - You can also create folders and import them
    - These are called "modules"
  - We usually put all code in a separate folder, e.g. `libs` or `utilities`
- Data, images and other assets should also be in their own folders

Improving Code

How not to get your peers angry

Debugging

- Hardest way: don't debug at all
- Easier: use `print()` statements at important places
- Best: use a debugger to trace the code execution
  - Every IDE (such as Visual Studio, VSCode, PyCharm, etc.) has one
- Most important concepts
  - Breakpoints
  - Step into, step over, step out
    - These usually have keyboard shortcuts assigned
  - Variable inspection
  - Interactive window; terminal
  - Call stack

Unit Testing

- Debugging and testing are very scientific processes
  - Intuitive for most people with math / science background
- Can show bugs in the code
  - Cannot show the code is bug-free!
  - "Absence of evidence is not evidence of absence"
- Unit tests: pieces of code that test other pieces of code
- Unit test layout: **AAA** (**A**rrange, **A**ct, **A**ssert)

```
def sum_numbers(a, b):
    return a + b
def test_sum_with_zeros():
    a = 0
    b = 0
    result = sum_numbers(a, b)
    assert result == 0

test_sum_with_zeros()
```

Other Types of Tests

- Unit testing ensures our functions work
- There are many more types of tests
  - Software: integration tests, regression tests, system tests, security tests, etc.
  - Data validation tests – these ensure correct formats of the data
- Statistical tests
  - Is my hypothesis (or data model) true?
  - Example: for linear regression $\Rightarrow R^2$
  - Another example: train / test set in machine learning
  - "Sanity checks"
  - Plotting graphs, comparisons, etc.
- It's absolutely important to check most (if not all) of our steps

Performance Tests

- Test how fast a code executes
  - Better: test the code complexity with different arguments
    - Possibly, plot the results
  - We can use the `time` library

```python
start = time.time()
for i in range(1000):
    sum(numbers)
stop = time.time()
print(stop - start)
```

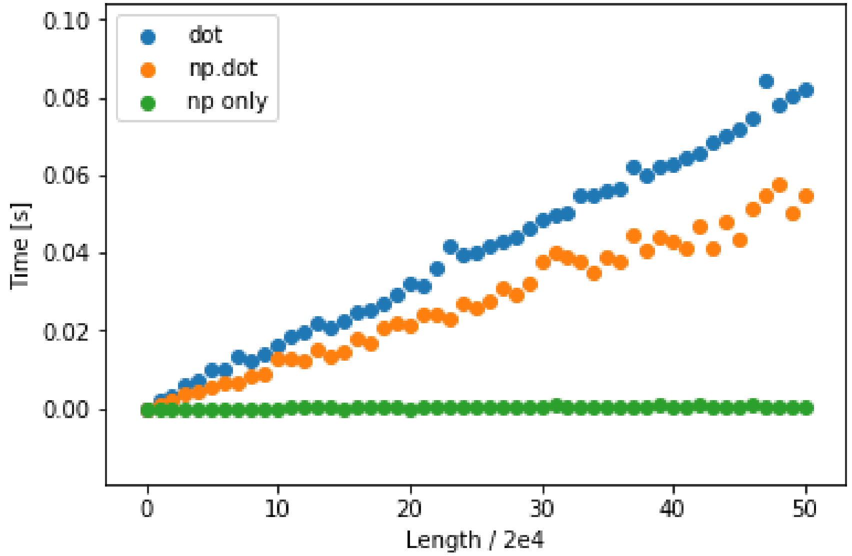
- Important: be careful how you check the time
  - Average execution over multiple trials to reduce random errors
  - Do not include initializations and "external" code
    - Code that you’re not interested in optimizing
- **Do not optimize prematurely!**

numpy **Performance**

- numpy is really fast on arrays and matrices
  - It works in C "behind the scenes"
  - Takes advantage of all elements being of the same type
- Vectorization – transforming the code so that it uses vectors and matrices

```
times = []
for test in range(51):
  a = np.random.uniform(1, 10000, size = test * 20000).tolist()
  b = np.random.uniform(1, 10000, size = test * 20000).tolist()
  start = time.time()
  dot(a, b) # Also: np.dot(); np.dot() directly
  stop = time.time()
  times.append((stop - start))
plt.scatter(range(len(times)), times)
```

- If possible, use numpy only
  - Avoid conversion to and from lists or other structures – this is slow



numpy **Performance (2)**

- Example: grayscale image from RGB
- 1140 x 550px
- The second block is **easier to write**, more intuitive, and **100x faster** (0,05s vs 0,5s on my machine)
- Correctness test

```
(new_img == np_img).all()
```

```
img = imread("...")
img_as_list = img.tolist()
start = time.time()
new_img = []
for row in range(len(img_as_list)):
  new_img.append([])
  for col in range(len(img_as_list[row])):
    new_img[row].append(0)
for row in range(len(img_as_list)):
  for col in range(len(img_as_list[row])):
    curr_sum = round(sum(img_as_list[row][col]) / 3)
    new_img[row][col] = curr_sum
stop = time.time()
print(stop - start)
```

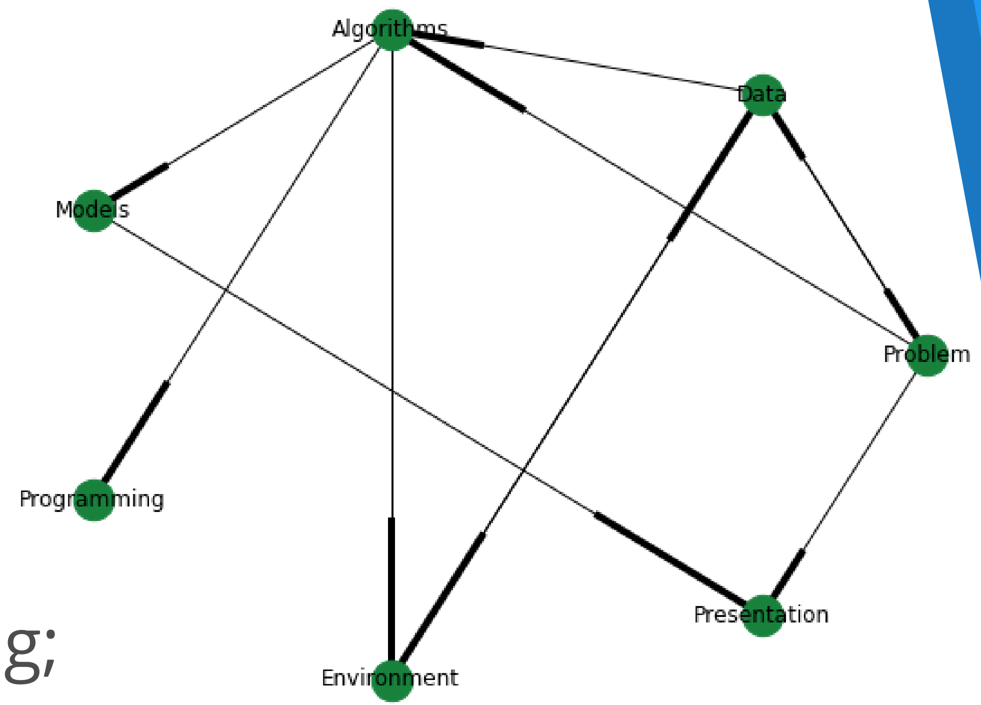
```
start = time.time()
np_img = img.mean(axis = 2).round().astype(np.uint8)
stop = time.time()
print(stop - start)
```

Reproducible Research

How to do stuff properly

Data Science Process

- As we know, the data science lifecycle is complex
  - For example, see this image
  - Also, there are a lot of topics
- Components and dependencies
  - Problem (task)
  - Data (experiments, dataset; different forms)
  - Algorithms (e.g. linear regression, Bayesian model)
  - Models (testing, selecting, fine-tuning; normalizing data, etc.)
  - Programming (APIs, functions, testing)
  - Environment (packages, such as numpy; or tools, such as Excel)
  - Presentation (tables, KPIs, visualizations, software)



Reproducible Research

- The whole process is complex, so we need a way to verify our work (and possibly other people's work)
  - Particularly important when a study affects decisions
  - Sometimes impossible: time, opportunity, money, etc.
- Why is it so important?
  - It’s **the only thing** we can guarantee about our study
- Easiest: supply all your data, and your notebooks
  - The notebooks contain all information about your research
- Requirements: analytical (not raw) data; code; documentation of the code and processes
- Markdown and LaTeX help us write more explicit documentation (in text and math format)

Reproducible Research (2)

- **Dos**
  - Good science (interesting, relevant problem; communication)
  - Automation of tasks (as many as possible)
  - Version control usage
    - Even if you're working alone, this helps you in the case something goes terribly wrong
  - Environment management (e.g. conda packages)
  - Sometimes: random seed, mock objects and other pseudorandom variables
- **Don'ts**
  - Manually edited data
    - If we get a new version, we have to edit the data again
  - Omitted (deleted) steps of the process
    - If a step you perform is not in your notebook, it can't be replicated easily

Comparing to Previous Work

- Both yours (if you have some) and others'
  - In the beginning: to see what others have done
  - In the end: to compare your findings to others'
- This can be a software product, or a paper, or something else
- Example: see papers at arXiv
  - Good examples of a scientific article layout
- Example 2: Kaggle notebooks (kernels)
- Don't forget to cite everyone that you've borrowed ideas, code, research methods, or information from
  - Reason 1: If they are proven wrong, your research may be wrong too
  - Reason 2: You're not plagiarizing them

Communicating Results

- Many possibilities
  - Sometimes, only an action to take: "discount product A by 40% next week to get an expected $50k $\pm$ 5k"
  - Other cases: dashboard (continuous analytics)
  - Deploying models to a production environment
    - If the model is passed data, it returns an output
  - Integrating into existing software
    - E.g. integrating a custom ad manager which recommends products to users
    - Or, deploy on Excel / PowerBI / Google Analytics / custom server-side script
  - Creating customer-facing software: not very common
  - Scientific paper (not too often, but depends)
- Be open to feedback
- Use a source control to track changes and share your work

Evidence-Based Analysis

- Once again, the entire process is very long
- At each step, we're making a lot of choices
  - Sometimes without even realizing it
  - E.g., default parameters in algorithms, default settings, **assumptions** about the data
- Main idea
  - Don't take these decisions randomly (or unknowingly)
    - Base them on previous research
  - This reduces the "degrees of freedom"
    - Therefore, accounts for better reproducibility
  - Also, guarantees that the used methods are widely accepted

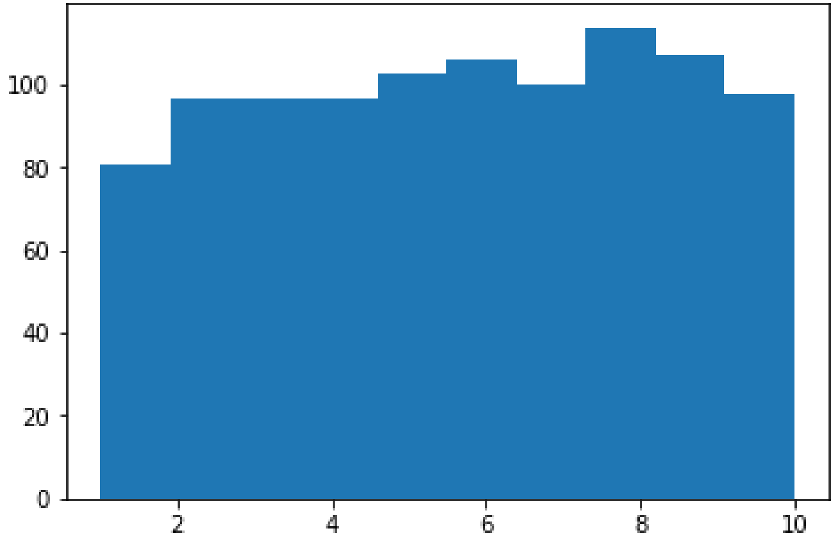
Evidence-Based Analysis (2)

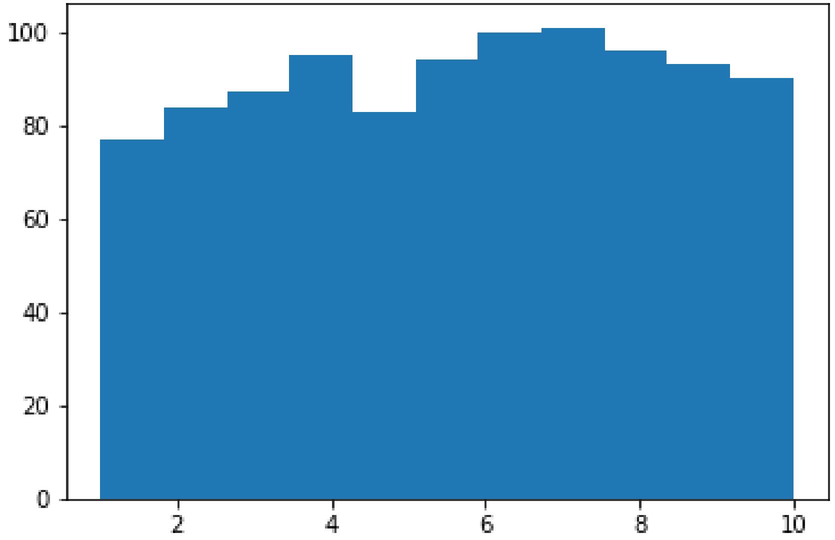
- Example
  - Default choice: 10 bins vs. [Freedman – Diaconis](#) bin size rule

```
np.random.seed(120)
x = np.random.uniform(
  1, 10, size = 1000)
```

```
plt.hist(x)
```

```
hist, bins = np.histogram(x, bins = "fd")
width = (bins[1] - bins[0])
center = (bins[:-1] + bins[1:]) / 2
plt.bar(center, hist,
    align = "center", width = width)
```





Last Note: Cognitive Biases

- No matter how good we are, we're all susceptible to biases in our reasoning
  - These can be used for good or bad
  - [List of biases]
    - Some popular ones: anchoring, choice support and "ostrich effect", confirmation bias, survivorship bias
  - We, as researchers, should try to overcome as many of these as possible
- This will also help find flaws in other people's methods
  - [How to Spot a Fake News Story]
  - [Three Ways to Spot Logical Fallacies]
  - [StatsDoneWrong website]

Summary

- High-quality code and software engineering best practices
  - Code conventions
- Data science project structure
- Improving code
  - Debugging, unit tests, performance tests
- Reproducible research
  - Tools, methods, ideas

Questions?