Designing & Implementing Data Pipelines for Scientific Research

Lecture 3:
Testing & Profiling Data Pipelines

Dr Ahmad Abu-Khazneh
Senior Machine Learning Engineer
Accelerate Programme
Spring School May 2023

ACCELERATE PROGRAMME FOR SCIENTIFIC DISCOVERY

SCHMIDT FUTURES

Verifying correctness of your pipeline

Test-Driven Development (TDD)

- TDD is one of the most prominent methodologies in software engineering for implementing bug-free software at scale.
- In my opinion it is also one of the more useful methodology to apply when implementing scientific data pipelines to ensure their correctness.
- Even if you don't adhere to the full methodology, some of the workflows that the TDD methodology promotes such as strict commitment to unit testing and its approach to refactoring are extremely powerful.
- Historically TDD emerged as a software methodology as a counter to another very prominent software methodology called the waterfall model.
- So before explaining how TDD works and how it can fit in your workflow it is useful first to briefly examine the waterfall model and its limitation.

The waterfall model

Description

- The waterfall model for developing software has been the dominant framework for implementing software until the beginning of the century.
- It roughly stipulates that software projects should be implemented in the following 4-steps process:

Requirement gathering —> Design —> Implementation —> Testing

- Note that this is a very natural way of implementing any project, not just software! Which also explains why it was so intuitive and dominant for a long period of time.
- In particular note that this is implicitly the process for which many scientists implement their code - even if they are not aware of following it.
- Even though it is quite intuitive, overtime this methodology became increasingly questioned by the software engineering community.

The waterfall model

Limitations

- Understanding the limitations of the waterfall model wasn't just an intellectual exercise - there was an extremely apparent trend of frequent large software projects failing - delayed, over budget, wrong implementation, …
- In the UK alone there are numerous high-profile public IT projects that failed even when outsourced to the best IT consultancies in the world.
- The frequency, magnitude, and diversity of IT projects failing led the software engineering community to the conclusion (after an enormous amount of research) that the failure must be in the process itself, it wasn't just in the way it was implemented.
- Can you suggest what are some of the limitations of the model from your own experience working on software projects in scientific research?

The waterfall model

Testing in the waterfall framework

- In particular software engineering researchers identified that one of the main flaws is that it leaves testing till the very end.
- On the face of it, keeping testing till the end seems like a natural thing to do: you first need to have something to test before testing it!
- However, withholding testing till the end of the project incentives all kinds of suboptimal workflows throughout the project, the two main ones being:
    - First, it means misinterpretations of the requirements are only uncovered right at the end after a lot of time has already been spent implementing code.
    - Second, it increases the chances of cutting corners on testing or reducing time spent on it since it is left to the end of the project close to the deadline.
- In general, by regulating testing till the end of the process the waterfall model de-emphasises its importance and separates it from the design and implementation process.

TDD methodology

- The TDD methodology turns the waterfall model on its head and proposes that testing should come first.
- Another important principle of TDD is that it provides very specific guidelines on what those tests should be and how they should be implemented.
- In particular TDD stipulates that testing should be done in the beginning of any project and written down before any other coding in the form of *unit tests*. A unit test is a test that clearly states what the output should be for a given input.
- Special consideration should be given to *edge cases.* Those are unit tests that specify what the output should be for troublesome cases such as boundary conditions so these have been empirically identified as a source of most run-time errors.

TDD methodology

Benefits of unit testing

- There are numerous benefits to writing the unit tests before any other code in the form of unit tests:
  - Unit tests act as a set of very clear specification for the code to be implemented, the programmers job is now to write code that passes the tests, so there is much less scope for misinterpretation
  - Since the unit tests have a simple structure (this input should return this output) they also act as a form of documentation for the codebase, i.e. they make the code *self-documenting*. This can be particularly useful for teams with significant churn in team members.
  - Unit tests also help you with *refactoring* code efficiently.

Refactoring code

- Code refactoring means restructuring the code without changing its external behaviour.
- In particular it usually involves the restructuring of the code to improve the way its designed, such as removing duplicate code snippets into their own function, or breaking down a big algorithm into a number of functions.
- The first version of your code is usually very messy: you are very focused on getting thing to run to verify your approach than on writing neat code.
- Hence refactoring is usually done at later stages of the development cycle once you have a substantial amount of code.

Refactoring code

Problems in refactoring code

- The main problem with refactoring is that it is difficult to change a substantial amount of the codebase while ensuring that nothing changes in its external behaviour.
- Even skilled programmers are weary of doing so which is why it is encouraged not to try and change legacy code if you can avoid it, even when it is tempting to do so since legacy code is usually bloated.
- The main worry is that refactoring code will break something in a subtle way, that it will introduce what software engineers call a *regression bug.*
- So another benefit of unit testing is that it enables you to refactor code without worry, since they will quickly identify if anything in the external behaviour of the code has broken after it has been refactored.

Limitations unit testing

- The main limitation of unit testing is that they require a lot of sunk cost in writing them.
- However the time spent in writing them is more than compensated by productivity gain such as being able to refactor efficiently and managing churn in your team (such as introducing new PhD students to work on the codebase after one of them leave)
- Another limitation is that unit tests only test that the code runs correctly, they don't test if the code runs efficiently. They treat your code as a blackbox.
- To test if your code runs efficiently you need to profile it.

Profiling code

- Profiling code means measuring the number of resources consumed by different components of your code.
- These resources include: computational resources, memory resources, network bandwidth, …
- The vast majority of scientific code is predominantly concerned in computational resources (and to a lesser extent memory) used which is usually directly proportional by the time spent running your code.
- Some data pipeline code, for example that make use of real-time data from sensors, might also be interested in profiling network bandwidth consumed.

Profiling code

- Profiling code is particularly useful in data pipelines because it allows you to identify bottlenecks in its flow which you can then think of ways of optimising.
- Profiling code usually entails profiling code on different volume, variety and velocity of input (so different datasets in the case of data pipelines)
- This might even lead to scientific insight, for example if you notice that certain datasets have very different profiles in terms of computational resources used when pipelined.
- There are many libraries and frameworks that are implemented to help you profile your code in most languages.
- In python: cProfile, timeit, Pyinstrument,… we will experiment with some of these in the lab.

Optimising pipeline code

- A logical step to attempt after profiling your pipeline and identifying bottlenecks is to experiment with ways of optimising it.
- This can be particularly useful as it will allow you to deal with more data in your research and can be highlighted as a new contribution of your pipeline if done successfully.
- However, the main problem is that optimised code is usually very sophisticated and this can cause many bugs to be introduced into your codebase.
- This highlights another benefit of unit tests. If you have comprehensive unit tests then you won't worry about introducing new errors into your codebase and thus can try and optimise it as much as you can.
- So even though Donald Knuth famously said "premature optimisation is the root of all evil." To highlight how easy it is introduce errors when focusing on optimisations, with unit tests you get *mature* optimisation!

TDD workflow

- So to summarise the TDD workflow:
    - Write unit tests
    - Check that unit tests fail correctly (important to verify they can pass trivially)
    - Implement code
    - Check code passes unit test
    - Refactor code
    - Check unit tests still pass
    - Profile code and identify bottlenecks
    - Optimise code if possible by reimplementing some steps
    - Check unit tests still pass

TDD workflow *in real life*

- While adhering to the strict tenants of TDD is rewarding in the long-term it is not always possible to do so in a research environment in academia due to time constraints, budget, team size….
- So my advise would be rather to try and selectively apply it to various parts of your project but not all of them.
- For example, it is a good idea to at least have comprehensive unit testing for at least the core parts of your data pipeline, which are usually the ones that perform the most sophisticated and mathematical operations in your pipeline.
- Another benefit of unit testing is that more and more journals have it on their checklist for submitted code so it is a good idea to have some unit testing for the most crucial parts of your submitted code to give people confidence in your submission.

Bonus slide

Unit testing and the future of programming

- One of the interesting recent developments in programming is the adoption of AI-assissted coding (such as copilot).
- And an interesting part of this trend is that those tools seem to be most useful when you provide the AI with very detailed specification of the code you intend to write.
- So this highlight an interesting emergent benefit of writing unit tests: they provide excellent specification for an AI-assissted coding tool.
- In other words, the future of programming might well consist of the programmer writing unit tests for which the AI then generates code that passes the tests!

Lab scenario

Unit testing and the future of programming

- To provide you with hands-on experience of TDD workflow in research team we will role-play the following common scenario.
- The PhD supervisor provides some high level description of bioinformatics code to be implemented (it will involve transcription of genomic sequences).
- PhD student 1 turn this description into python unit tests using the PyTest framework.
- PhD student 2 writes code that passes the unit test and identifies an edge case that was missing from the unit tests.
- PhD student 1 then updates the unit tests to clarify the edge case situation.
- PhD student 3 profiles the code and identifies bottleneck in PhD student's 2 code
- PhD student 4 who joins the team after a while figures out a clever way to refactor the code and optimise it, but finds out that her optimisation breaks one of the unit tests because of the obscure edge case she forgot to account for.