Delivering robust Machine Learning pipelines

Eduardo Blancas Reyes

Data Scientist, Fidelity Investments

November, 12, 2019. Northeastern University.

Motivation

- Your job as a Data Scientist is to *create value* (e.g. optimize resources, increase customer satisfaction, etc) from data
- Since you do not know what will work in advance, a lot of work will depend on trying out things (e.g. focus on a specific subset of customers, add features from a new dataset), yout success is directly related to how many experiments you try
- The data you use will never be in the right format so you need to code the necessary transformations until it does, we call this a "pipeline" – *this concept applies to any data processing project, but we will focus on ML applications*
- Faulty ML pipelines severely delay experiments, or even worse, they will make you reach the wrong conclusion
- A robust data pipeline that reduces the opportunity for failures and prevents errors from propagating to your model is an essential component for any Data Scientist

Examples of problems with faulty pipelines

- Lack of reproducibility: your current best model achieves great performance in the validation set but your coworker is unable to get such performance with the "same" code
- Degraded performance goes unnoticed: your model used to have 90% precision but it is now at 70%, you have made so many changes that you do not know which one produced the performance drop
- Debugging hell: you spent an entire week hunting down the bug, it was a line of code dropping an important feature from your 100+ total features

ML systems are software systems – but are not developed like them (1/2)

- ML systems are software systems, however, common good practices in traditional software engineering have not found their way into ML software engineering
- One of such practices is modularity: software systems are divided in smaller units that interact with each other through an interface, this is a desirable property that allows us to test components independently in a predictable way
- Another such practice is how to incorporate improvements in the codebase: **small changes** are **automatically tested** in a **clean environment** and only merged after all tests pass and someone reviews code changes

ML systems are software systems – but are not developed like them (2/2)

- **Small changes.** Make code reviewing easier (imagine going through a 1,000+ lines of code change)
- **Automatic tests.** Any change is a risk for new bugs, the only efficient way to ensure that the project still works as expected is to run a series of tests on every code change
- **Clean environment.** The final model will not run on your laptop (or the development server), it is important to make sure the project as a whole works in a clean environment

Software Engineering workflow example

- SE1: Hey did you finish working on the payment service?
- SE2: Not yet, I just have to add some input validation logic
- SE1: Ok let me know when it's done
- SE2: *finishes writing the validating function and a few tests*, *pushes to the repo*, *all tests pass*, *someone reviews code*
- SE1: *pulls code*, *starts his/her developing environment*, *interfaces the payment service with the email confirmation service*

(broken) ML workflow example (1/2)

- DS1: Hey, are you done cleaning dataset X?
- DS2: Not yet, I am still fixing a few things
- DS1: Ok, let me know when it's done so I can use it in the model
- DS2 *opens up Jupyter*, *keeps working on the code to clean the data*, *applies to code to the raw dataset*, *saves the data in* `/data/ds1/clean/dataset.parquet`, and pushes Jupyter notebook to git
- DS2: Hey, I just finished, clean data is in `/data/ds1/clean/dataset.parquet` and the notebook is in the repo
- DS1: Great, thanks. **loads data from** `/data/ds1/clean/dataset.parquet`, *runs a function for feature generation* and *trains a model*

(broken) ML workflow example (2/2)

- DS1 used a dataset generated by DS2 without actually reviewing or testing the code
- Aside from obvious downsides due to lack of code review and testing, there are other consequences specific to ML projects
- **Lack of reproducibility**: there is no guarantee that running DS2's notebook will generate the exact same output that DS1 used, reproducubility is *paramount* in ML systems
- **Breaking changes**: any steps that take clean dataset X as input (directly or indirectly) might break due to the introduced changes (e.g. if they expect columns to be named in a certain way)

If it is so bad, why does it happen?

- In traditional software, progress can be objectively measured (e.g. user can log in, user can add things to their shopping cart, etc.), but it is harder to measure it in ML projects
- Some success metrics might be defined (say, we need at least 80% precision and 70% recall)
- This metric-based evaluation makes Data Scientists rush to get close to those numbers as soon as possible
- With an overemphasis in metrics, good engineering practices are completely overlooked since they take time an effort that is not perceived to contribute to this goal

Good engineering practices translate in faster progress

- The key is to strike a good balance between modeling (creating new features, optimizing hyperparameters, try other models) and engineering (simplifying code, writing tests)
- It is important to mention that these engineering practices are language-agnostic (no matter if you use Python or R) nor depend in any external library, **they just depend on giving proper structure to our project**

Improved ML workflow

- DS1: Hey, are you done working on cleaning dataset X?
- DS2: Not yet, I am still fixing a few things
- DS1: Ok, let me know when it's done so I can use those in the model
- DS2: *modifies code for cleaning dataset X*, *pushes to the repo*. Hey, I just finished
- DS1: *reviews code*, *tests pass*, *changes are merged*, *DS1 uses results generated from running DS2's code*

Properties of robust pipelines (1/2)

1. Pipeline are naturally composed of small tasks, make this explicit in the source code (**Modular**)
    - Modularity allows us to test our pipelines in a granular way (test tasks as opposed to the whole pipeline)
    - It makes easier to modify the pipeline (to delete and add tasks)
2. Build purely functional tasks, the only way to modify a task should be through its inputs (**Stateless**)
    - State is one of the most common sources for software bugs
    - The most common source of state is using global variables (never, ever use them) and unnecessary object-oriented programming

Properties of robust pipelines (2/2)

3. Tasks are easily accessible from a script, for example, if you are building a Python pipeline, should be easy to import (**Discoverable**)
    - We cannot automate the end-to-end pipeline execution if tasks are not accessible from a single script
4. Dependencies among tasks are clear from the source code (**Structured**)
    - The source code alone should be enough to understand the pipeline workflow (run this first, then this, then this)
    - Using documentation for this is a bad idea – no one will keep it up-to-date
5. The pipeline should not make any assumptions about storage resources (i.e. the filesystem) or available libraries for it to easily run in a new environment (**Portable**)
    - Put it another way: do not hardcode paths to files and add a script to install dependencies

What could go wrong in a stateful pipeline?

```
class DataCleaningTask:
    def __init__(self, parameter)
        self.parameter = parameter

    def remove_bad_quality_columns(self, data):
        # 20+ of lines code here...
        pass

   def impute_nas(self, data):
        # another 20+ lines of code here...
        pass

    def run(self, raw_data):
        data = self.remove_bad_quality_columns(raw_data)
        data = self.impute_nas(data)
        return result

def make_features_task(clean_data):
    # do more processing
    return result

raw_data = load_raw_data()
data_cleaning_task = DataCleaningTask(parameter=1)
clean_data =  data_cleaning_task.run(raw_data)
features = make_features_task(clean_data)
```

Sample pipeline (1/2)

```
# tasks.py

# Modular and Stateless: tasks are pure functions and there is no shared state
# This is a Python example, but your code could be anything (e.g. SQL script),
# the idea is to avoid any kind of "global" or "shared" state

def download_A(output_path):
    # get data from the network and save it in output path
    pass

def download_B(output_path):
    # get data from the network and save it in output path
    pass

def merge(path_to_A, path_to_B, output_path):
    # merge the two datasets and save it in output_path
    pass
```

Sample pipeline (2/2)

```
# pipeline.py

# Discoverable: tasks are easily importable
from tasks import download_A, download_B, merge
from config import paths

# Portable: the code makes no assumptions about the filesystem
# file locations are parameters that can easily changed
# Structured: it is clear that merge depends on A and B
download_A(output_path=paths['A'])
download_B(output_path=paths['B'])

merge(paths['A'], paths['B'], output_path=paths['merge'])
```

Improvement #1: Design to scale

- As your project evolves, performance might become an issue (e.g. you incorporate new datasets, add a long-running feature engineering script, running out of memory)
- While you should not worry (too much) about performance until it becomes an issue, taking some measures will delay this and help you be prepared
- The key consideration is to separate the *what to achieve* from the *how to achieve it*
- This translates into prefering *declarative programming over imperative programming* (e.g. SQL over imperative-style Python)
- Declarative programming scales much easier (you can scale up a SQL script with no code changes by just using a bigger server)

Improvement #2: Faster experiments

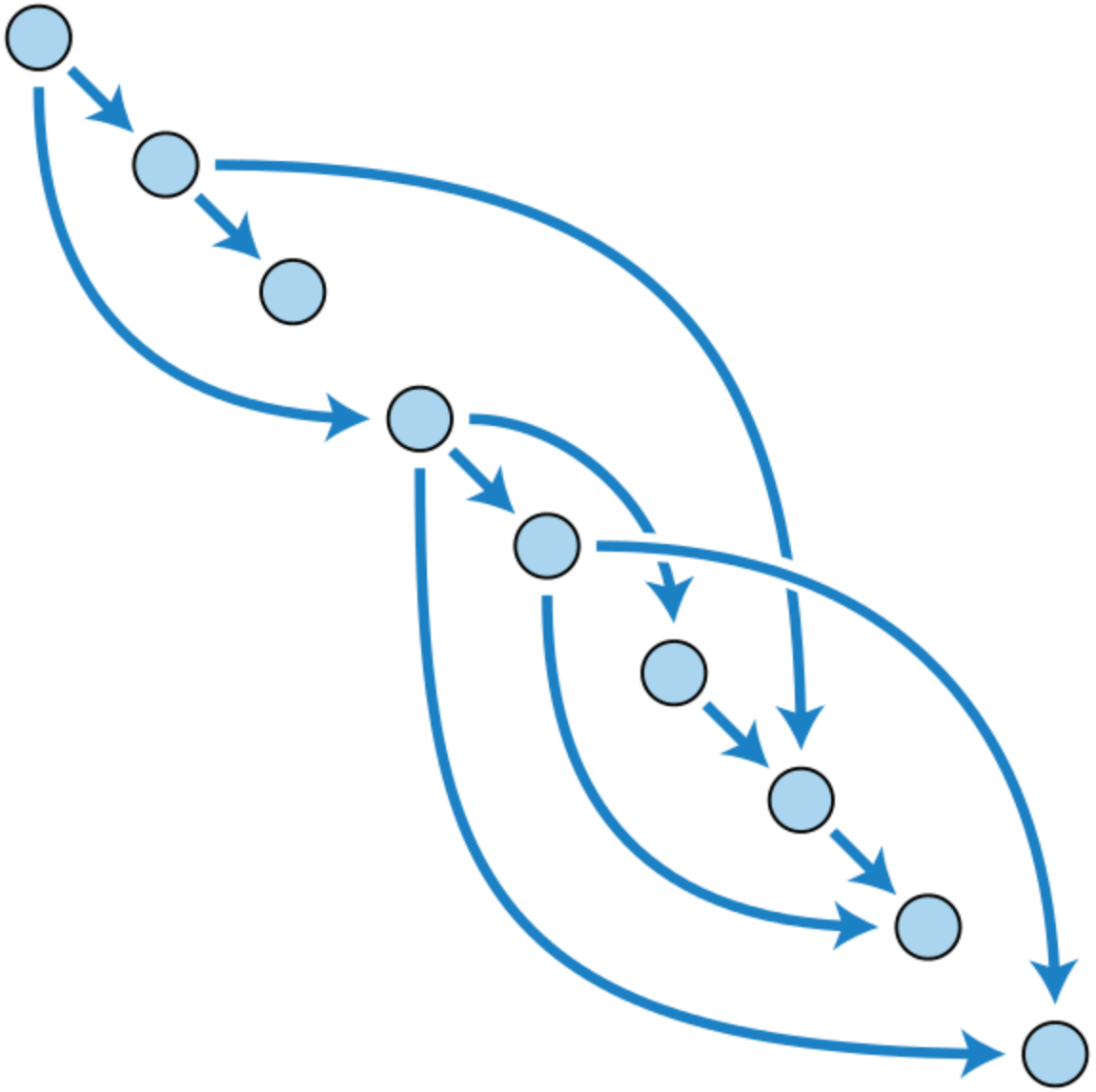
- The previous workflow has a big problem: it is terribly slow
- Imagine you have a pipeline with 20 tasks and your colleague modifies one line of code in the very first task
- Doing an end-to-end run will trigger executing the 20 tasks, ML code usually takes hours (or even days) to run, not a good idea
- But we should just run tasks affected by the new change
- This is exactly what software build systems (such as GNU Make)
- You can use GNU Make for your projects, the main caveat is that GNU Make expects all outputs to exist locally, which is often not the case (e.g. a table in a remote database system)

A (simple) pipeline building system (1/4)

- When should you re-run a task? When its output is expected to change
- Given the pipeline's stateless architecture, only two things can change any task's output: their inputs or its code
- But their inputs are a consequence of its dependencies code
- This translates in running a task whenever its code or any of its dependencies code has changed

A (simple) pipeline building system (2/4)

- A data pipeline is a DAG: a directed graph with no cycles
- We can sort the DAG so that every task is run after all its dependencies are done
- The algorithm is called topological sorting



Source: Wikipedia

A (simple) pipeline building system (3/4)

```
# assume you have a "pipeline object" which is represented
# as a directed graph and has no cycles
for task in topological_sorting(pipeline):
    if dependencies_ran(task) or task_code_changed(task):
        run(task)
```

A (simple) pipeline building system (4/4)

- If all your processing happens locally GNU Make is an option (there are several tutorials online)
- There are some libraries with similar objectives but they have the same limitation as GNU Make (assume all happens locally)
- On the other hand, current workflow management tools are more *data engineering* oriented and AFAIK, do not provide incremental builds (Airflow, Luigi, etc.)

Improvement #3: Testing (1/4)

- In traditional software: tests usually run for a short amount of time (seconds) against a deterministic and simple output
- Example:

```
from testing import Mailbox
from my_online_store import Customer, Product

def test_shopping_receipt_is_sent():
    mail = Mailbox(email='customer@domain.com')

    customer = Customer(email='customer@domain.com')
    product = Product(uid=123)
    customer.add_to_cart(product)
    customer.buy()

    assert mail.get_latest_message().subject == 'Your receipt from online store'
```

Improvement #3: Testing (2/4)

- Those three facts do not hold for data pipelines:
  - Tasks usually take a lot to run (minutes or even hours)
  - The output is not simple (a dataset, a model)
  - Some tasks have randomness involved (training a model)
- To tackle long-running tasks, you can use a pipeline building system to avoid running unnecessary computations or run your pipeline in a data sample (say a 10% random sample – although this will introduce even more randomness)
- For complex outputs, it is often challenging to find what to test, fortunately, the most common errors are very easy to detect (missing columns, missing values, sudden drops in performance)

Improvement #3: Testing (3/4)

- Tackling randonmess is a bit harder but keep things simple: setting the random seed in your tests is helpful, also, you can test that some value falls within certain small range instead of a specific value – if your results vary a lot between runs, there is probably something wrong
- Some advice on testing:
  - Write acceptance tests for each task, make and end-to-end run on upon codebase changes
  - Use logging (Python's `logging` module is great!), it will help you debug failing tests
  - As your project evolves, improve testing coverage

Improvement #3: Testing (4/4)

```
import logging
import pandas as pd
from utils import load_model

def clean_customers_data(path_to_customers_data):
    # code for cleaning data
    logger = logging.getLogger(__name__)
    logger.info('Dropping %i customers due to missing customer ID', n_drop)

def test_clean_customers_data(path_to_clean_customers_data):
    df = pd.read_parquet(path_to_clean_customers_data)
    assert not df.customer_id.isna().sum(), 'Customer ID cannot be null'

def train_model(path_to_clean_customers_data):
    # train model and store model + metrics
    pass

def test_train_model(path_to_model):
    model = load_model(path_to_model)
    # test both sides, sudden increases in performance
    # could also be a bug!
    assert 0.80 <= model['metrics']['accuracy'] <= 0.81
```

Summary

- Striking a good balance between modeling and good software engineering practices are key for a ML project to succeed
- A robust data pipeline has 5 properties: Modular, Stateless, Discoverable, Structured and Portable
- The properties are language-agnostic, they only depend on giving proper structure to our project
- Design to scale: prefer declarative over imperative programming
- Use a pipeline building system to accelerate end-to-end runs
- Write acceptance tests upon task execution

Comments? Questions?

Twitter: @edublancas