Designing & Implementing Data Pipelines for Scientific Research

Lecture 2:
Designing Data Pipelines

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

ACCELERATE PROGRAMME FOR SCIENTIFIC DISCOVERY

SCHMIDT FUTURES

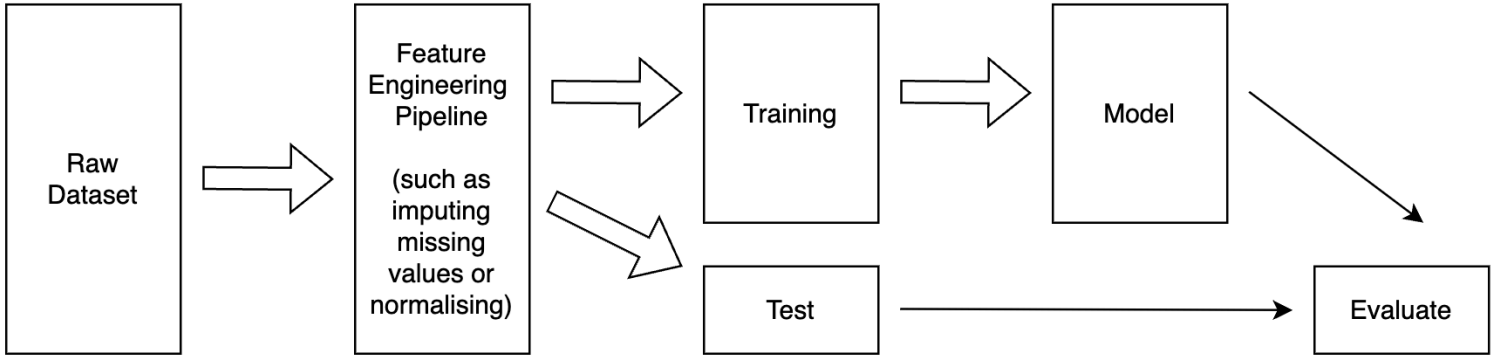
Good pipeline design

- In this lecture we want to explore what are good practice to follow when designing data pipelines for scientific research
- As mentioned previously there are no firm theory or algorithm to follow in designing pipelines but there are some good general principles and heuristics
- Different heuristics make sense in particular domains and we will highlight this contrast in the lab when exploring your own pipelines since you each come from a different scientific discipline.
- Sometimes using frameworks and libraries specifically designed to help scaffold pipelines could provide good design off-the-shelf for many use case.
- So for this lecture we will focus on Sklearn's pipeline object as it provides good template on how to design pipelines.

Properties of good pipeline design

Reproducibility

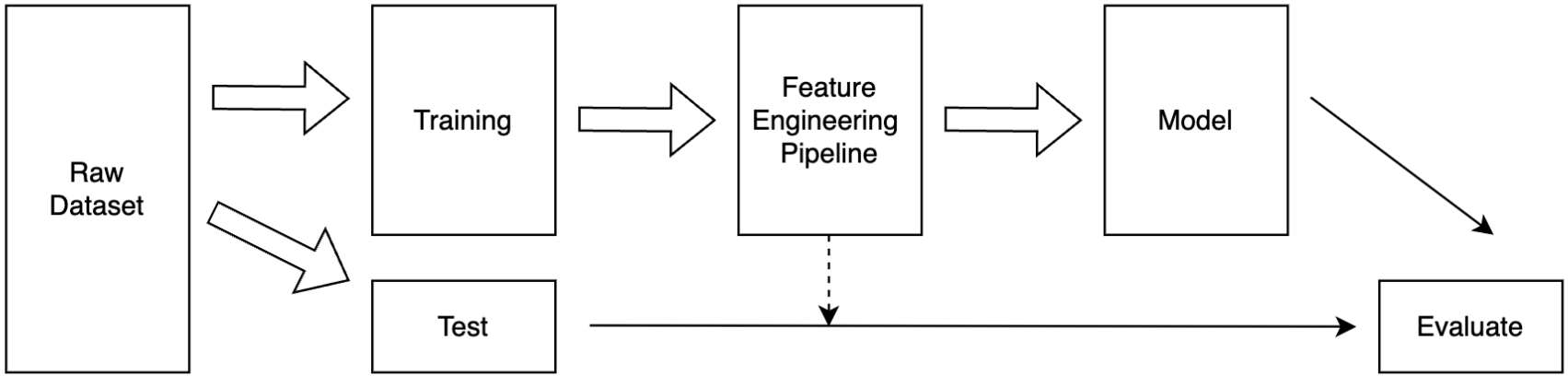
- The first general rule is that pipelines should be easy to reproduce. This means with minimal friction you can rerun every step of a pipeline on different data and without triggering any other action (or side effects) beyond the pipeline.
- This might sound like an obvious rule but much scientific code neglects it.
- To achieve it requires conscious effort by the programmer to separate pipelining code from other code, which is rarely done given the fallacy we spoke about previously of some scientist not thinking of the pipeline as an entity unto itself.
- For example, a lot of scientific code closely blends the pipelining code with the modelling code, making it difficult to only run the pipelining without the modelling.
- The consequence of making your pipeline hard to reproduce in the way stated above can sometimes even lead to methodological flaws in your model, meaning you can generate wrong model evaluations scores! Let's demonstrate this in the following high-level example.


Raw Dataset
Feature Engineering Pipeline
(such as imputing missing values or normalising)
Training
Test
Model
Evaluate

Methodological flaws

The problem

- The previous diagram shows a common ML pipeline set-up, but it does contain an error. This might be surprising as this setup is widely used including in some high-ranking kaggle entires.
- Can you guess where the methodological flaw is?
- The flaw is that the feature engineering pipeline is applied before splitting the dataset. The reason is that some of the feature engineering steps such as normalising and imputation allows some information from the test dataset to “leak” into the training dataset (can you see why?)
- The following slide shows how to fix this error.



The pipeline is *calibrated* on the training dataset,
and the *calibrated version* is run again on the test dataset

Methodological flaws

The fix

- We see that the correct way to setup your ML pipelineis to run the data pipeline twice, once on the training dataset so as to calibrate it, then you take the calibrated version and run it on the test dataset.
- We note here that one of the reasons why this flaw is very common is because implementing a pipeline in a way that is cumbersome to run, makes running it once the path of least resistance so unconsciously we try to do so.
- Moral: bad design can lead to bad methodology, design is not just for aesthetics!

Properties of good pipeline design

Reproducibility

- The previous general example proves one of the benefits of reproducibility, but the main benefit of reproducibility is of course much more important and obvious: you need to make your pipeline as easy to replicate by others so that a) they can verify it b) they can reuse it on other datasets.
- Making your pipeline easy to run makes it also easy to experimenting with by toggling different steps on and off to see what effects they have on your model which can yield very interesting insight. So another benefit is that it allows you to gain productivity boost when experimenting with it in this way.

Properties of good pipeline design

Auditability

- Auditing is very common in the finance world, investors need to be able to look at your accounts to inspect all the biases in your accounting procedure to gain confidence in your bottom line.
- Auditing is of course implicitly very important in research - you should make it easy for others to observe every single step you carried out on your data.
- In particular, many of the steps that are usually carried out in pipelines can induce a lot of opaque biases and cover subjective decisions that can make a big difference in the aggregate to your ‘bottom line’ (usually your model accuracy)
- Some of these subjective decisions: how did you deal with outliers? How did you deal with missing data? Which columns did you throw away? Which new features did you introduce?
- Scientists must make it easy for others to note all these decisions that take place in the data pipeline.

Properties of good pipeline design

Auditability

- Many scientific programmers might think it is sufficient to provide source code with your paper to achieve auditability.
- This of course provides *some* auditability but it is more in keeping with the letter of the law of auditability rather than spirit of it
- The problem here is that if your code quality is poor and your pipeline is not structured in a way that makes it easy for others to inspect where every step begins and ends, then you are not really providing auditability.
- We will see in the lab how sklearn makes it easy for pipelining code to attain auditability by explicitly forcing you to provide a clear manifest of each step of the pipeline and their sequencing.
- Moreover, there is a more profound reason why providing source code alone is arguably not sufficient to attain full auditability. This is best explained after the next property of good design.

Properties of good pipeline design

Modularisability

- This is fortunately one of the software engineering principles that are highlighted in many introductory data science courses.
- However, it is still mis-implemented in many scientific code, in particular in the pipelining code.
- This manifests itself for example in that a lot of scientific code makes it hard for others to run the pipeline by excluding or changing the parameters of some of the steps in the pipeline.
- Having a uniform interface for each step can help achieve modularisabilty - you want your steps to be like LEGO bricks, they each have a well defined interface that can easily plug into another brick.

Properties of good pipeline design

Modularisability

- Again, sklearn help you achieve this by providing you with an interface that you can use to define your own steps, and which sklearn itself uses to define numerous popular feature engineering steps.
- Circling back to how modularisability help attain auditability in your pipeline: strong auditability require that no only is it important to see what decisions have taken place in a data pipeline, but also being able to easily inspect the effect of such decisions.
- Making your pipeline more modular means that others can easily take out or slightly modify different steps and inspect the impact on the rest of the pipeline.

Data pipelines in Scikit-learn

The pipeline object

- The pipeline object in scikit-learn is designed to meet the aforementioned desirable properties of pipelines, so it's a good idea to use it when implementing pipelines in python.
- Even if you don’t use it, it’s useful to study it and use it as a template when implementing your own pipeline from scratch (or in another language).
- A pipeline object in scikit is simply a chain of scikit transformer objects, so let’s define what those are first.

Data pipelines in Scikit-learn

Transformers

- Transformer objects in sickit are simply objects that transform an input X in some way,
- So they need to contain a fit(X,y) method that doesn't return a value, just simply store the learnt data inside the object.
- This might be surprising to some of you as we usually associate the fit method with the model implementation stage (such as fitting the coefficients of a logistic regression), but even simple feature engineering steps are fitted in some sense.
- For example imputing values by using the most frequent value is a fitted step in the sense that it depends on what the most frequency value is in a column.
- Transformers also contain a transform(X) function that does change X based on the stored learn data set with the fit() function.
- So you first calibrate it with some values according a training dataset using fit, then use the calibrated version to transform another dataset using transform X

Data pipelines in Scikit-learn

Transformers

- Scikit learn also comes with a lot of transformers that you would want to use in your pipeline for feature engineering.
- For example it has many common imputers in its sklearn.impute package and many common statistical normalisers (such as Box-Cox) in its sklearn.preprocessing.
- Moreover an important point is that you can easily implement your own transformer to adhere to the Sickit protocol by extending the BaseEstimator class (which we will see in the labs)
- This means you have to define your own fit() and transform() function for your transformer.
- This also means that others can use a particular transformer that you implemented even if they are not interested in the rest of your pipeline, which is an example in how using sklearn's pipeline can nudge you into best-practice design.
- In fact their is a thriving eco system of third-party sklearn transformer some carrying out some very sophisticated feature engineering steps.

Data pipelines in Scikit-learn

Pipeline and transformers

- Going back to the Pipeline object, it is simply defined as a list of (key, value) pairs, where the key is a string describing the step, and the value refers to a transformer object.
- Thus this instantly help attain one of the desirable properties of a pipeline design - auditability. Someone can easily inspect what takes place inside your pipeline by inspecting this list of key value pairs that provides a manifest of the pipeline steps. They can then peek into a particular transformer if they want to see how it in turn was implemented.
- This is much neater than presenting your pipeline as one monolithic coding block that overlaps with other part of your codebase.
- Moreover, a pipeline comes with its transform() and fit() function which when they are invoked they in turn call the fit and transform functions of each of the steps defined in the pipeline (since they are transformers they much have their own version of these methods).
- This helps attain another of the desirable properties of good pipeline design which is to make pipeline easy to run.
- It also help attain another desirable property: by commenting out some of the steps in the pipeline definition the invoking fit and transform again, you can easily experiment on what impact does each step have on the final output of the pipeline or even the evaluation of the final model.

Pipeline Design lab

- For the pipeline design we will first go through a published pipeline that makes use of the sklearn pipeline object to get more insight on how it can be used.
- We will then focus on rewriting parts of your pipeline to fit into the sklearn pipeline object, we will first assess how far it strays from the best-practice that you gain when using the sklearn pipeline. After scaffolding your pipeline with the sklearn pipeline object it will then be useful to gain some experience in rewriting some of your steps as custom transformer objects.