An example of a complex pipeline using Prefect 2 (Orion)

# Introduction

## Overview

This article offers an introduction to pipeline management using Prefect2. This software has recently been launched in experimental mode but has the potential to be a step change over other pipeline management tools.

For illustration I will use an existing application which has a complex pipeline with multiple steps, nested loops, the need for warm restarts and many other features. It has been running successfully using Luigi for four years.

The focus of this article is on the pipeline management aspects of the project rather than the data science.

## Pipeline Management

There are a number of objectives for a good pipeline management tool:

1. Separate the pipeline management from core code that does the heavy lifting
2. Execute tasks in parallel potentially on different machines
3. Provide a UI to monitor all the jobs, flows and tasks
4. Restart pipelines without rerunning successful tasks
5. Cope with normal programming control structures such as loops, nested loops and conditional execution.
6. Addition of new tasks dynamically during the flow

There are dozens of pipeline management tools available. None meet all of the above objectives.

* They tend to use a two step process of creating a DAG then executing. This makes it harder to debug and to create tasks dynamically without jumping through hoops.
* They are generally quite intrusive requiring code to be rewritten in a specific way so the pipeline management is dictating the whole project rather than a separate component
* The core of pipeline management should be point 5 above. Yet it seems that for most pipeline management packages these control structures require a completely new syntax.

An iniital look at prefect2 suggests it used normal python code to implement loops and control structures; has no DAG; and is unintrusive to existing code.

# Target application

The target application creates an electronic record of who voted at an election. This is useful to help prioritise election campaigns. If we have limited resources then we want to focus on those who voted in the last five elections above those that were eligible but never voted. Surprisingly this information is only available on paper.

The electoral register is a database of all registered voters. At elections this is printed out and each polling station manually checks off electors as they arrive on these paper copies. This creates what is known as the “marked register”.

After the election these marked register pages are scanned into electronic image files. Typically each consituency has 100K voters split across 100 polling stations. The marked register for a constituency consists of around 100 pdfs of 10 pages and 100 voters per page. This is repeated across 600 constituencies. An example of the paper sheets is shown below:

EXAMPLE OF IMAGE

Text

Description automatically generatedWe have working functions that extract the data we want from the scanned pages and match it to the database. These apply a wide range of data science techniques such as OCR, clustering, deep learning, text vectorizing and sequence alignment. However these will not be covered here as this article is focused on pipeline manaagement

* Each function will be applied multiple times to different pdfs, pages and voters
* Much of this is embarassingly parallel
* It needs to be robust to failure. If one page fails then we don’t want to reject the pdf. If one pdf fails then we still want to process all the other pdfs.

WITH LIGHT THEMEt

class Page:

    def \_\_init\_\_(self, path):

        self.path = path

        self.pandoc = self.read()

        self.meta = self.getmeta()

# Jobflow without pipeline management

The data for a single job includes a csv with a list of voters and a set of pdfs with image scans representing the same voters.

The code below shows the steps to extract the data for all the pages ready for feeding into the model:

def jobflow(jobid):

downloaded = pretasks.download(jobid)

csv = pretasks.create\_csv()

all\_matched = []

    for pdf in glob("\*.pdf"):

       pdf\_pages = pdftasks.get\_pages(pdf)

       for png in pdf\_pages:

            rotated = pngtasks.rotate(png, config)

            boxes = pngtasks.OCR(rotated, config)

            parsed = pngtasks.parse(boxes)

            matched = pngtasks.match(parsed, csv)

            all\_matched.append(matched)

return all\_matched, csv

This is easy to read but each page will be processed in sequence on a single processor. If we have 8 processors then it would be better to process eight pages in parallel then to merge it all together at the end.

Also if there is any failure the job will fail. One unreadable page should not make the whole job fail. If we have 20K pages there will almost certainly be some failures. Rather than failing we want to log problems and measure the output. If 99% of pages are processed that may be good enough. Alternatively we might decide to investigate failures in which case we want to rerun only the tasks that have failed not those for the 19,800 pages that succeeded.

# Implementing the target application in Prefect2

## Quick start

To adapt the above code for prefect2 is really simple:

* Decorate the jobflow function with @flow(task\_runner=DaskTaskRunner())
* Decorate the pngtasks with an @task decorator

Now when jobflow is run it sets up a dask cluster; and any @task functions in the @flow function are submitted to the dask cluster and return a future. The future is like a reference to the task. This allows us to find out the state of the task and to obtain the result once it is complete.

Dependencies between tasks are handled automatically by passing the futures as parameters. The downstream task will wait until their input futures complete before executing. This means there is no need to separately define links between tasks.

Prefect2 has enabled parallelisation and resilience to failure with minimal change to the code.

## Configuring dask

Typically no configuration is necessary. There is a dask dashboard on port 8787 that shows detailed information on cpu and memory usage by worker.

In this case the dask dashboard shows only 4 workers. The reason is that prefect sets the default number of workers to the number of physical CPUs. My laptop has 8 vCPUs so it may be faster to set n\_workers=8. If I do this it then the log shows a number of memory usage warnings. The reason for that is that memory is divided equally between the workers and 1/8 of the memory is not sufficent for some tasks. Therefore finally I set n\_workers=6.

## Adding pretasks

There are several tasks that need to be executed sequentially at the start of the job. Whilst these cannot run in parallel there are benefits to adding these to the managed pipeline. This would provide a unified view of task failures and timings; make it easier to restart; and easier to adapt.

Just adding the @task decorator these tasks would be carried out in parallel. Unlike the pngtasks there is no data passed between them to indicate dependencies so they won’t wait automatically. Instead we use the special parameter wait\_for e.g. create\_csv(wait\_for=[download]); or call the wait method e.g. download.wait().

In the case of get\_pages we also want to iterate over the result directly in the flow. The wait method returns a State object so we need to call get\_pages().wait().result() to get the list of pages. After maiking all these changes the code looks like this:

@flow(task\_runner=DaskTaskRunner(cluster\_kwargs=dict(n\_workers=6)))

def jobflow(jobid):

    downloaded = pretasks.download(jobid)

    downloaded.wait()

    csv = pretasks.create\_csv().wait().result()

    all\_matched = []

    for pdf in glob("\*.pdf"):

        pdf\_pages = pdftasks.get\_pages(pdf).wait().result()

        pdf\_pages = pdf\_pages():

        for i, png in enumerate(pdf\_pages):

            rotated = pngtasks.rotate(png, config)

            boxes = pngtasks.OCR(rotated, config)

            parsed = pngtasks.parse(boxes)

            matched = pngtasks.match(parsed, csv)

            all\_matched.append(matched)

return all\_matched, csv

## Adding posttasks

The jobflow above returns all\_matched (list of futures) and csv (future). The tasks after this will merge all\_matched into a single dataframe and then merge it into the csv..

The merging steps are primarily carried out in sequence not in parallel. At the same time they are processing more data so require more memory. The Dask cluster would not be appropriate as we would only be using one of the CPUs and have access to 1/6 of the memory. Using the SequentialTaskRunner() executes these on a single processor with 100% of the memory.

The “predict” step uses machine learning to make predictions. Pytorch uses MKL to utilise multiple processors for prediction so does not require Dask. Trying to use MKL and Dask to manage multiprocessing would lead to conflicts.

For these reasons the posttasks are put in a separate flow using the SequentialTaskRunner:

@flow(task\_runner=SequentialTaskRunner())

def mergeflow(s3path, matched, csv):

matched = [v.wait().result() for v in matched]

    merged = posttasks.merge(\*matched)

    csvonly\_out = posttasks.csvonly(merged, csv)

    data = posttasks.prepare\_data(merged)

    preds = posttasks.predict(data)

    output = posttasks.create\_output(csv, merged, preds)

    posttasks.upload(s3path, wait\_for=[output, csvonly\_out])

# Comparison with Luigi

In luigi to create a task you need to create a Task class; incorporate the function into the run method; override other methods as required; and create a Parameter class for each parameter. The whole project has to be structured for luigi. By contrast in prefect2 you just add the @task decorator. No changes are required to code. This makes it simpler and also enables unit testing independently of pipeline management.

In luigi there is a DAG. You can have a loop inside a task which creates new tasks using the yield statement but this is a hack that feels awkward and is hard to read later. In prefect2 there is no DAG and you can use a python loop that creates new tasks at runtime.

Luigi runs everything together on one machine. In prefect2 the scheduler and UI run in a separate process, container or machine; there is a range of flow runners such as kubernetes or docker; and mutliple task runners for execution including Dask and Ray.

Overall prefect2 uses a much more straightforward approach that meets most of the goals of pipeline management.

# Conclusion

This article has shown an example of how Prefect2 can be used to create a complex pipeline.

A pipeline manager’s main purpose is to manage all the standard control flows with conditions and loops. This functionality sounds a lot like the definition of “programming” so it is surprising that most pipeline tools invent their own syntax. Prefect2 uses standard python which is refreshing. It also stays out of the way. The pipeline code is completely separate to core functionality.

There are still some issues. Prefect2 is still “experimental”. There are bugs. There are holes in the documentation. There are some things that could be improved. However the core functionality is working and it can only improve. Likely will be production ready later this year.

There is one feature required for this project that is not handled by prefect2 and that is for the pipeline to work similarly to makefile, saving files at each stage, launching tasks only if they are not already complete. This is not part of prefect2 nor on the roadmap. Prefect2 does have caching but this seems opaque and over-complex. Hopefully it will improve.

Until that happens perhaps we can extend prefect2? I will cover this in part 2.

Extending prefect2 to implement makefile like functionality

# Introduction

This article shows how to extend prefect2 to implement makefile functionality. It is also illustrative of the extent to which prefect2 can be adapted to other bespoke requirements.

As outlined in the previous article my application processes a large number of images of text to extract information. This requires going through a number of steps to transform each image.

When the application was developed it was easy to get it working with data from a single town. However each town generates data independently. Whilst the aim is the same the data can vary in a myriad of ways. We have no control over the quality or format of their data nor do we have a representative sample that covers all the possible variations.

This made development iterative. Each new town generated some failures which were fixed as discovered. There could be thousands of pages that work then a few that don’t. For each iteration we want to only run the pages and tasks that previously failed without repeating those that were successful already.

# Prefect2 way of working

Prefect works quite well for defining flows and handling restarts. It can also cache results in the file system. So at first glance it should handle a makefile like approach well but…

## Prefect works top down

It starts with the first task in each flow. So even if the task does not need to be run it gets sent to the scheduler and is reported as cached. This may be a small amount of unnecessary work but it becomes an issue with thousands of tasks. If I have 100K tasks and 1 failed then I only want the one that failed to run.

Luigi works bottom up. It looks at the upstream dependencies and if they are available already the task will run. If they are not available it will move upstream to run what is needed only.

## Prefect cache is hidden

Prefect2 provides the name for the cache file but this cannot be user defined nor templated for specific inputs. The name of the cache file is stored in the database but in a particularly cryptic way and is not easily accessible. You cannot easily view the cache nor delete parts of the cache to force a rerun. The only way to force a rerun is delete the whole cache otherwise if the cached file exists then it will be used.

If I carry out OCR on image1 I want to store the result as OCR/image1. Then I can easily use file explorer to see what has been completed and view the output for a specific step/image. I can also force a task to rerun by deleting the appropriate output files.

Sometimes I want the task to store the output rather than returning it for storing.

# Adapting prefect2 for makefile

## Why not just do it a different way?

This might be possible. However:

1. I am porting an existing application and want to replicate the outputs exactly to fit with existing tools and to enable easy testing and comparison of outputs.
2. My development approach has worked well for a number of similar projects. I don’t see an alternative that would work as well with the type of projects I do.
3. Regardless It is useful to see how easy it is to adapt Prefect2 because no package meets all needs all of the time

## Extra parameters for @task

I want to provide a parameter for @task to add a target output path. This will be a template that can be filled at execution time using context variables such as taskname and inputs to the task.

Would be useful for the template to act like a python formatstring i.e. include code. Any functions would need to be defined at runtime so it will be a limited list e.g. os.path, datetime.

To enable this to be filled at runtime means overriding Task.\_\_init\_\_ and Task.\_\_call\_\_. In this way we can intercept calls where output is complete and bypass the scheduler.

Skipped tasks return a Skipped object. The skipped object implement methods such as wait and result but is not a Future. These are converted into data when used as parameters to another task.

## Autosaving and autoloading files

When a task returns data I want this to be saved and replaced by a path. When this path is used a a parameter I want it loaded as data.

It is useful for debugging to save interim outputs. This allows you to iterate over selected outputs to check it is as expected. Where outputs are large it is preferable to avoid keeping them in memory or passing them between processes and machines. In these cases better to pass paths to data rather than data itself.

# Appendix

FOCUS ON PIPELINE ONLY

There are a number of issues which add to the complexity:

* Pdfs
  + Can be one pdf for 100 polling stations or two pdfs for 1 polling station
  + We have no control over the source data. It is what it is.
  + There are multiple systems used by different areas with variations in format
* Pages
  + Quality of scan can vary. Sometimes very poor using 1960s scanner
  + Can be rotated 90/180. Can have smaller rotations and creases.
  + Some pages contain no voters or are blank
  + Can contain 1, 2 or 3 block columns of voters
* Voters
  + The columns of data are nor distinct and can run into each other
  + Many are crossed out by the system because they are not eligible to vote
  + Some are crossed out to indicate they voted
  + Crossed out letters cannot be read perfectly by OCR
  + Crossings out make any distinction between data columns difficult
* Matching
  + A voter can be on the electronic record but not on the printed page; or vice versa. This is due to timing differences.