Snakemake 

“A scalable bioinformatics workflow engine”

Vukan Jevtić

February 25, 2020

Technische Universität Dortmund

Motivation: Analysis workflow automation

We would like to automate our analysis workflow

Make

+ Universal, always available
+ Supports abstraction
– Exclusively Bash
– Hard to read
– Hard to debug
– Hard to find rule for specific file

  Especially if abstraction is used

Motivation: Analysis workflow automation

We would like to automate our analysis workflow

Make

+ Universal, always available
+ Supports abstraction
– Exclusively Bash
– Hard to read
– Hard to debug
– Hard to find rule for specific file

  Especially if abstraction is used

Snakemake

+ Install via conda
+ Recipes can contain either Bash or Python(!)
+ Recipes are named
+ Much easier to read
+ Abstractions are easy to understand
+ Can submit jobs to a computing cluster
+ If rule fails: Corrupted output is deleted
– Additional `.snakemake` directory

Make abstraction example

In every analysis, some degree of abstraction is needed

The following Makefile generates a plot for each dataset in current directory

```
interp=python
files:=$(shell find . -name '*.csv')
plots:=$(patsubst %.csv,%.pdf,$(files))

all: ${plots}

%.pdf: %.py %.csv
    ${interp} $< $@ $(word 2, $^)
```

This is already quite hard to read

Snakemake installation

```
$ conda config --add channels bioconda
$ conda install snakemake
```

Cite as:

- Köster, Johannes and Rahmann, Sven. “Snakemake - A scalable bioinformatics workflow engine”. Bioinformatics 2012.

Further reading (links)

- Documentation
- Implementation details
- Even more implementation details (TU Dortmund thesis)

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Make

```
plot.pdf: data.csv
    python plot.py

data.csv: raw_data.csv
    python selection.py
```

Minimal console command

```
$ make
```

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Minimal console command

```
$ snakemake
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Make

```
plot.pdf: data.csv
    python plot.py

data.csv: raw_data.csv
    python selection.py
```

Running by specifying output file

```
$ make plot.pdf
```

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Running by specifying output file

```
$ snakemake plot.pdf
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Make

```
plot.pdf: data.csv
    python plot.py

data.csv: raw_data.csv
    python selection.py
```

Running by specifying name of rule

```
$ ???
```

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Running by specifying name of rule

```
$ snakemake make_plot
```

Anatomy of a Snakefile

Workflow: `raw_data.csv` → `data.csv` → `plot.pdf`

Make

```
plot.pdf: data.csv
    python script.py

data.csv: raw_data.csv
    python selection.py
```

Snakemake

```
rule make_plot:
    input:  "data.csv"
    output: "plot.pdf"
    shell:  "python plot.py"

rule select_data:
    input:  "raw_data.csv"
    output: "data.csv"
    shell:  "python selection.py"
```

Output Input Recipe

+Self-Documentation

“snakemake -nr” console output (dry run with reason for each rule)

```
Building DAG of jobs...
Job counts:
        count   jobs
        1       make_plot
        1       select_data
        2

[Wed Feb 19 15:50:05 2020]
rule select_data:
    input: raw_data.csv
    output: data.csv
    jobid: 1
    reason: Missing output files: data.csv


[Wed Feb 19 15:50:05 2020]
rule make_plot:
    input: data.csv
    output: plot.pdf
    jobid: 0
    reason: Missing output files: plot.pdf; Input files updated by another job: data.csv

Job counts:
        count   jobs
        1       make_plot
        1       select_data
        2
This was a dry-run (flag -n). The order of jobs does not reflect the order of execution.
```

“snakemake” console output

```
Building DAG of jobs...
Using shell: /usr/local/bin/bash
Provided cores: 256
Rules claiming more threads will be scaled down.
Job counts:
        count   jobs
        1       make_plot
        1       select_data
        2

[Wed Feb 19 15:38:12 2020]
rule select_data:
    input: raw_data.csv
    output: data.csv
    jobid: 1

[Wed Feb 19 15:38:12 2020]
Finished job 1.
1 of 2 steps (50%) done

[Wed Feb 19 15:38:12 2020]
rule make_plot:
    input: data.csv
    output: plot.pdf
    jobid: 0

[Wed Feb 19 15:38:14 2020]
Finished job 0.
2 of 2 steps (100%) done
Complete log: /net/nfshome/home/somepath/.snakemake/log/2020-02-19T153812.433826.snakemake.log
```

Physics analysis example

To train a BDT, we usually need a selected data and mc file

```
rule data_preselection:
    input:  "data_raw.root"
    output: "data_selected.root"
    shell:  "python selection.py data_raw.root data_selected.root"

rule mc_preselection:
    input:  "mc_raw.root"
    output: "mc_selected.root"
    shell: "python selection.py mc_raw.root mc_selected.root"

rule train_bdt:
    input:
      "data_selected.root",
      "mc_selected.root"
    output: "bdt.model"
    shell: "python train_bdt.py data_selected.root mc_selected.root"
```

`input`, `output`, `shell` etc. are optional

Physics analysis example

To train a BDT, we usually need a selected data and mc file

```
rule data_preselection:
    input:  "data_raw.root"
    output: "data_selected.root"
    shell:  "python selection.py data_raw.root data_selected.root"

rule mc_preselection:
    input:  "mc_raw.root"
    output: "mc_selected.root"
    shell: "python selection.py mc_raw.root mc_selected.root"

rule train_bdt:
    input:
      "data_selected.root",
      "mc_selected.root"
    output: "bdt.model"
    shell: "python train_bdt.py data_selected.root mc_selected.root"
```

Would be nice to reduce amount of repetitions

Physics analysis example

We can alias files ⇒ rules can reference their own parameters

```
rule data_preselection:
    input:  "data_raw.root"
    output: "data_selected.root"
    shell:  "python selection.py {input} {output}"

rule mc_preselection:
    input:  "mc_raw.root"
    output: "mc_selected.root"
    shell: "python selection.py {input} {output}"

rule train_bdt:
    input:
      data = rules.data_preselection.output,
      mc   = rules.mc_preselection.output
    output: "bdt.model"
    shell: "python train_bdt.py {input.data} {input.mc}"
```

Strings containing `{...}` are formatted

Executing arbitrary python code in Snakefiles

A Snakefile can be treated almost like a python script:

```
import uproot
import pandas
import numpy as np

def say_hello(name):
    print(f"Hello {name}!")

rule somerule:
    input: files = [f"dataset_{num}.root" for num in range(100)]
    run:
        say_hello("E5")
        for tfile in input.files:
            ds = uproot.open(tfile)["DecayTree"]
            data = ds.arrays("B_P[XY]", outputtype=pandas.DataFrame)
            print(np.sqrt(data.B_PX**2 + data.B_PY**2))
```

Executing python scripts

Instead of `shell` or `run` a script can be invoked.
(It does not need to be a python script)

```
rule massfit:
    input: "data.root"
    output: "parameters.txt", "plot.pdf"
    params:
      fitConstrained = False,
      extendedMLFit = True
    script: "massfit.py"
```

massfit.py:

```
import ROOT as R
from ROOT import RooFit
fitContrained = snakemake.params.fitConstrained
extendedMLFit = snakemake.params.extendedMLFit
# Load datasets, fit something...
```

Useful command line options

Just print scheduled rules without running

```
$ snakemake <rule> -n
```

Print the reason for running each rule as well

```
$ snakemake <rule> -n -r
```

Force execution of target

```
$ snakemake <rule> -f
```

Force execution of a target and its workflow

```
$ snakemake <rule> -F
```

Force re-execution of rule and its workflow

```
$ snakemake <rule> -R
```

Run workflow until specified rule

```
$ snakemake <rule> --until <rule>
```

Update timestamps →force files up to date

```
$ snakemake <rule> --touch
```

Ignore errors

```
$ snakemake <rule> --keep-going
```

Rerun incomplete rules (in case of crash)

```
$ snakemake --rerun-incomplete
```

Print shell commands that snakemake runs

```
$ snakemake -p
```

expand(...)

Snakemake has an integrated method for generating lists of files: `expand(...)`

```
1 rule file_requester:
2     input: expand("file_{cat}_{num}.txt", cat=["A", "B"], num=range(3))
```

The following list is created as input:

`file_A_0.txt, file_A_1.txt, file_A_2.txt, file_B_0.txt, file_B_1.txt, file_B_2.txt`

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
rule single_selection:
    input:  "data_{num}.root"
    output: "data_{num}_selected.root"
    shell:  "python run_selection.py {input} {output}"

rule select_files:
    input: expand("data_{n}_selected.root", n=range(10))
```

Note:

1. Input and output *must* contain same wildcards
2. A wildcard rule cannot be called directly by its name
3. Two rules should not contain the same outputs

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
1 rule single_selection:
2     input:  "data_{num}.root"
3     output: "data_{num}_selected.root"
4     shell:  "python run_selection.py {input} {output}"
5
6 rule select_files:
7     input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake select_files
```

rule `select_files` is going to call the wildcard rule for 10 different files

Next level of abstraction: Wildcards

A wildcard rule matches patterns in dependencies

```
1 rule single_selection:
2     input:  "data_{num}.root"
3     output: "data_{num}_selected.root"
4     shell:  "python run_selection.py {input} {output}"
5
6 rule select_files:
7     input: expand("data_{n}_selected.root", n=range(10))
```

If one runs

```
$ snakemake data_7_selected.root
```

rule `select_files` is going to call the wildcard rule for case `num = 7`

The `wildcards` object

Inside a wildcard rule, a variable named "`wildcards`" is defined

```
rule my_wildcard_rule:
    input: "file_{channel}_{polarity}_{year}.root"
    output: "file_{channel}_{polarity}_{year}_out.root"
    message: "Reading {wildcards.year} file"
    run:
        print("Running the rule for year = {wildcards.year}")
        if wildcards.year == "2017" and wildcards.channel == "B2JpsiKstar":
            print("This is my favourite dataset!")
        shell("python run_selection {input} {output} -year {wildcards.year}")
```

But what if a certain combination of wildcards needs to be treated differently?

$\Rightarrow$ Use wildcard constraints

Wildcard constraints

```
rule somerule:
    input:  "data_{year}.root"
    output: "massplot_{year}.pdf"
    wildcard_constraints: year="201[578]"
    shell: "python massfit.py {input}"

rule somerule_special_case:
    input:  rules.somerule.input
    output: rules.somerule.output
    wildcard_constraints: year="2016"
    shell: "python massfit.py {input} -be_careful"
```

If you need to treat a wildcard value differently from the others, you need to constrain them for each relevant rule as shown here
Here, regex can be quite useful: regex101.com.

Advanced Topics

Virtualisation in Snakemake via Singularity

Single rules (or the whole Snakefile) can be configured to run in an arbitrary virtual environment

```
rule envrule:
    input: "file.txt"
    output: "outfile.txt"
    singularity: "/path/to/singularity/container.simg"
    shell: "SomeShellCommand"
```

This is limited to `shell` and `script` execution

When calling snakemake, singularity needs to be activated:

```
$ snakemake envrule --use-singularity --singularity-args
            "--bind /run,/ceph,/net"
```

Binding `/run` is obligatory, the rest is optional

When `singularity: ...` is defined outside of a rule it is implied for all rules

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

But what do you do if you need $\mathcal{O}(100)$ CPUs, for example 300 CPUs and 1TB of RAM?
⇒ Send your jobs to our own cluster! (For large jobs, ask for permission first!)

Parallelizing everything!

At some point, you may realize that you need more than 1 CPU...
Luckily, there is an option for that:

```
$ snakemake my_analysis -j20
```

This command is going to (try to) parallelize your workflow into 20 parallel chains

But what do you do if you need $\mathcal{O}(100)$ CPUs, for example 300 CPUs and 1TB of RAM?
⇒ Send your jobs to our own cluster! (For large jobs, ask for permission first!)
But what do you do if you need $\mathcal{O}(1000)$ CPUs?
⇒ Send your jobs to the new LiDO cluster of our university

Using snakemake to submit to a computing cluster

A tutorial can be found here: `Click me!`
Submitting to a computing cluster can be as simple as:

```
$ snakemake <rule> -j999 --drmaa
```

Using snakemake to submit to a computing cluster

A tutorial can be found here: `Click me!`
Submitting to a computing cluster can be as simple as:

```
$ snakemake <rule> -j999 --drmaa
```

...but usually you need to inherit environment variables from the host machine

```
$ snakemake <rule> -j999 --drmaa $' getenv=True'
```

Remember the space before the first parameter

Using snakemake to submit to a computing cluster

A tutorial can be found here: `Click me!`
Submitting to a computing cluster can be as simple as:

```
$ snakemake <rule> -j999 --drmaa
```

...but usually you need to inherit environment variables from the host machine

```
$ snakemake <rule> -j999 --drmaa $' getenv=True'
```

Remember the space before the first parameter

It would be even nicer, if snakemake could determine the amount of resources for each job...

Letting Snakemake manage job resources during cluster submission

A `cluster.json` file is recommended for defining a default configuration

cluster.json

```
{
  "__default__" : {
    "log" : "{rule}_{wildcards}.log",
    "err" : "{rule}_{wildcards}.err",
    "RAM" : "1GB",
  },
  "targets" : {
    "RAM" : "100MB"
  }
}
```

Letting Snakemake manage job resources during cluster submission

In Snakefile, define the amount of threads for each rule, otherwise default is used:

```
rule clusterrule:
    input: "file.txt"
    output: "outfile.txt"
    threads: 12
    run:
        print(f"This rule is allowed to use {threads} threads")
```

Then submit via

```
$ snakemake -j999 --cluster-config cluster.json --drmaa
            $' getenv=True \n output = {cluster.log} \n
            error = {cluster.error} \n
            request_memory = {cluster.RAM} \n
            request_cpus = {threads}'
```

And lean back while the cluster takes off 🚀

Subworkflows

Snakefiles can be connected via subworkflows:

Main Snakefile

```
subworkflow another_worklow:
    workdir: 'path/to/other/workdir'
    snakefile: 'path/to/other/workdir/Snakefile'

rule master_rule:
    input: another_worklow("text.txt")
```

Another Snakefile

```
rule create_file:
    output: "text.txt"
    shell: "touch text.txt"
```

Some more useful tips

Various file wrappers:

- Timestamp of files wrapped in `ancient("filename")` is ignored
- Files wrapped in `protected("filename")` are not deleted by Snakemake
- A file wrapped in `temp("filename")` is deleted after rule is finished
- `touch("filename")` creates an empty file with that name as output

Setting a function as rule input:

```
1 def get_files(wildcards):
2     return #[ A list of files according to wildcards]
3
4 rule arule:
5     input: get_files
```

Some more useful tips

Various file wrappers:

- Timestamp of files wrapped in `ancient("filename")` is ignored
- Files wrapped in `protected("filename")` are not deleted by Snakemake
- A file wrapped in `temp("filename")` is deleted after rule is finished
- `touch("filename")` creates an empty file with that name as output

Using a config file

config.json

```
{
  "param_a" : "362",
  "param_b" : "cat"
}
```

Snakefile

```
configfile: "config.json"

param_a = config["param_a"]
param_b = config["param_b"]
```

Common Errors

Common mistakes: Wrong wildcard deduction

```
1 rule somerule:
2     output: "afile_{year}_{polarity}.root"
3     shell: "echo Running rule"
4
5 rule requester:
6     input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`

Common mistakes: Wrong wildcard deduction

```
rule somerule:
    output: "afile_{year}_{polarity}.root"
    shell: "echo Running rule"

rule requester:
    input: "afile_2017_MagnetUp_garbage.root"
```

This is valid code: rule requester is calling somerule with (for example) `year="2017_MagnetUp"` and `polarity="garbage"`
This will eventually lead to an error →define what wildcard values are allowed

```
wildcard_constraints:
    year="201[5678]",
    polarity="Magnet(Up|Down)"
```

Note: these are regex strings