

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

```
1 interp=python
2 files:=$(shell find . -name '*.csv')
3 plots:=$(patsubst %.csv,%.pdf,$(files))
4
5 all: ${plots}
6
7 %.pdf: %.py %.csv
8     ${interp} $< $@ $(word 2, $^)
```

This is already quite hard to read

```
$ 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

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Minimal console command

```
$ make
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

Minimal console command

```
$ snakemake
```

Anatomy of a Snakefile

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

Make

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Running by specifying output file

```
$ make plot.pdf
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     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

```
1 plot.pdf: data.csv
2     python plot.py
3
4 data.csv: raw_data.csv
5     python selection.py
```

Running by specifying name of rule

```
$ ???
```

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     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

```
1 plot.pdf: data.csv
2   python script.py
3
4 data.csv: raw_data.csv
5   python selection.py
```

Output

Input

Recipe

Snakemake

```
1 rule make_plot:
2     input: "data.csv"
3     output: "plot.pdf"
4     shell: "python plot.py"
5
6 rule select_data:
7     input: "raw_data.csv"
8     output: "data.csv"
9     shell: "python selection.py"
```

+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

```
1 rule data_preselection:
2     input:  "data_raw.root"
3     output: "data_selected.root"
4     shell:  "python selection.py data_raw.root data_selected.root"
5
6 rule mc_preselection:
7     input:  "mc_raw.root"
8     output: "mc_selected.root"
9     shell:  "python selection.py mc_raw.root mc_selected.root"
10
11 rule train_bdt:
12     input:
13         "data_selected.root",
14         "mc_selected.root"
15     output: "bdt.model"
16     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

```
1 rule data_preselection:
2     input: "data_raw.root"
3     output: "data_selected.root"
4     shell: "python selection.py data_raw.root data_selected.root"
5
6 rule mc_preselection:
7     input: "mc_raw.root"
8     output: "mc_selected.root"
9     shell: "python selection.py mc_raw.root mc_selected.root"
10
11 rule train_bdt:
12     input:
13         "data_selected.root",
14         "mc_selected.root"
15     output: "bdt.model"
16     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 \Rightarrow rules can reference their own parameters

```
1 rule data_preselection:
2     input: "data_raw.root"
3     output: "data_selected.root"
4     shell: "python selection.py {input} {output}"
5
6 rule mc_preselection:
7     input: "mc_raw.root"
8     output: "mc_selected.root"
9     shell: "python selection.py {input} {output}"
10
11 rule train_bdt:
12     input:
13         data = rules.data_preselection.output,
14         mc   = rules.mc_preselection.output
15     output: "bdt.model"
16     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:

```
1 import uproot
2 import pandas
3 import numpy as np
4
5 def say_hello(name):
6     print(f"Hello {name}!")
7
8 rule somerule:
9     input: files = [f"dataset_{num}.root" for num in range(100)]
10    run:
11        say_hello("E5")
12        for tfile in input.files:
13            ds = uproot.open(tfile)["DecayTree"]
14            data = ds.arrays("B_P[XY]", outputtype=pandas.DataFrame)
15            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)

```
1 rule massfit:
2     input: "data.root"
3     output: "parameters.txt", "plot.pdf"
4     params:
5         fitConstrained = False,
6         extendedMLFit = True
7     script: "massfit.py"
```

massfit.py:

```
1 import ROOT as R
2 from ROOT import RooFit
3 fitConstrained = snakemake.params.fitConstrained
4 extendedMLFit = snakemake.params.extendedMLFit
5 # 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

```
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))
```

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

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

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

⇒ Use wildcard constraints

Wildcard constraints

```
1 rule somerule:
2     input: "data_{year}.root"
3     output: "massplot_{year}.pdf"
4     wildcard_constraints: year="201[578]"
5     shell: "python massfit.py {input}"
6
7 rule somerule_special_case:
8     input: rules.somerule.input
9     output: rules.somerule.output
10    wildcard_constraints: year="2016"
11    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.

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**

ADVANCED TOPICS

Virtualisation in Snakemake via Singularity

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

```
1 rule envrule:
2     input: "file.txt"
3     output: "outfile.txt"
4     singularity: "/path/to/singularity/container.simg"
5     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

Using snakemake to submit to a computing cluster

A tutorial can be found here: [Click me!](#)

Submitting to a HTCondor computing cluster can be as simple as:

```
$ snakemake <rule> -j999 --profile htcondor
```

Setting optional resource requirements for the scheduler

```
1 rule clusterrule:
2     input: "file.txt"
3     output: "outfile.txt"
4     threads: 12
5     resources:
6         MaxRunHours=24,      # Job takes up to a day
7         request_memory=1024 # Request RAM in MB
8         request_gpus=1,     # Submit to a machine with GPU
9         request_disk=1000000 # Disk requirement in kB
10    run:
11        print(f"This rule is allowed to use {threads} threads")
```

Submit with the same command lean back while the cluster takes off 🚀

Subworkflows

Snakefiles can be connected via subworkflows:

Main Snakefile

```
1 subworkflow another_worklow:  
2     workdir: 'path/to/other/workdir'  
3     snakefile: 'path/to/other/workdir/Snakefile'  
4  
5 rule master_rule:  
6     input: another_worklow("text.txt")
```

Another Snakefile

```
1 rule create_file:  
2     output: "text.txt"  
3     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

```
1 {
2   "param_a" : "362",
3   "param_b" : "cat"
4 }
```

Snakefile

```
1 configfile: "config.json"
2
3 param_a = config["param_a"]
4 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

```
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"`

This will eventually lead to an error → define what wildcard values are allowed

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

Note: these are regex strings