lts-workflows Documentation

Release 0.2.4+1.gc0fa646.dirty

Per Unneberg

Apr 05, 2018

Contents

The lts-workflow module is a top-level module for workflow repositories developed by the SciLifeLab Bioinformatics Long-term Support team.

Note that this module doesn't actually contain any workflows. Rather, it provides helper functions and documentation of a more general nature that relates to all workflows.

Contents:

CHAPTER 1

Installation

1.1 Stable release

To install lts-workflows, run this command in your terminal:

> **Warning:** WIP: as of yet there is no scilifelab-lts channel.

```
$ # conda install -c scilifelab-lts lts-workflows
$ conda install -c percyfal lts-workflows
```

This is the preferred method to install lts-workflows, as it will always install the most recent stable release.

1.2 From sources

The sources for lts-workflows can be downloaded from the Bitbucket repo.

You can either clone the public repository:

```
$ git clone git@bitbucket.org:scilifelab-lts/lts-workflows.git
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

CHAPTER 2

Configuration guide

The configuration guide provides a basic overview of the general options and settings layout for the supported workflow managers. Please refer to the workflow documentation pages for more specifice instructions regarding a particular workflow.

Currently, the supported workflow managers are:

- snakemake: a workflow management system written in python
- nextflow: a fluent DSL for data-driven computational pipelines

2.1 Snakemake

Snakemake can be configured through a configuration file that is passed either via the `--configfile` command line option, or the Snakefile `configfile:` directive. Once loaded, the configuration settings can be accessed through the global python object `config`.

Internally, configuration objects are python dictionaries, where the keys correspond to configuration options. This has the unfortunate consequence that it is difficult to provide a documentation API to the options. This text tries to address this issue, albeit in an unsufficient manner. As a last resort, for now at least, one simply has to look at the source code to get an idea of what the options do. In most cases though, the key names themselves should give an idea of what behaviour they target.

It is important to keep in mind that no validation of user-supplied configuration files is done. Consequently, should the user supply a non-defined configuration key, it will passed unnoticed by Snakemake. This can be frustrating when debugging; you are sure that you have changed a configuration value, only to notice later that the configuration key was misspelled.

2.1.1 Implementation

The configuration is constructed as a hierarchy of at most three levels:[1]

[1] Note that the configuration structure can vary depending on workflow since different developers work on different workflows. The structure described in this document was developed for the first iteration of the workflows.

```
section:
  subsection:
    option:
```

The `section` level corresponds to an application, or a configuration group of more general nature. The `subsection` can either be a new configuration grouping, or an option to be set. For applications, the `subsection` often corresponds to a given rule. Finally, at the `option` level, an option is set.

2.1.2 Configuration sections

For each workflow, there is a subdirectory named `rules`. The directory contains rules organized by directories and `settings` files that provide default configuration values. Every rule directory has its own settings file. There are two top-level settings file located directly in the `rules` directory, namely `main.settings` and `ngs.settings`.

settings

The `settings` section defines configurations of a general nature.

```
settings:
  sampleinfo: sampleinfo.csv
  email: # email
  java:
     java_mem: 8g
     java_tmpdir: /tmp
  runfmt: "{SM}/{SM}_{PU}"
  samplefmt: "{SM}/{SM}"
  threads: 8
  temporary_rules:
    - picard_merge_sam
```

For all settings, see `rules/main.settings`.

Importantly, many of these settings are *inherited* by the application rules, so that changing `threads` to 4 in `settings`, will set the number of threads for all configurations that inherit this option. However, you can fine-tune the behaviour of the inheriting rules to override the value in `settings`; see *Application settings*.

Here, the most important option is `sampleinfo`, which **must** be set. The `runfmt` and `samplefmt` options describe how the data is organized. They represent python miniformat strings, where the entries correspond to columns in the sampleinfo file; hence, in this case, the column **SM** and **PU** must be present in the sampleinfo file. So, given the following sampleinfo file

```
SM,PU,DT,fastq
s1,AAABBB11XX,010101,s1_AAABBB11XX_010101_1.fastq.gz
s1,AAABBB11XX,010101,s1_AAABBB11XX_010101_2.fastq.gz
s1,AAABBB22XX,020202,s1_AAABBB22XX_020202_1.fastq.gz
s1,AAABBB22XX,020202,s1_AAABBB22XX_020202_2.fastq.gz
```

samplefmt will be formatted as `s1/s1` and runfmt as `s1/s1_AAABBB11XX` or `s1/s1_AAABBB22XX`, depending on the run. The formatted strings are used in the workflows as *prefixes* to identify targets. Rules that operate on the runfmt will be prefixed by `s1/s1_AAABBB11XX` or `s1/s1_AAABBB22XX`, rules that operate on the sample level (i.e. after merging) will be prefixed by `s1/s1`.

Currently, the tests define three different sample organizations.

```
sample:
  runfmt: "{SM}/{SM}_{PU}_{DT}"
  samplefmt: "{SM}/{SM}"
sample_run:
  runfmt: "{SM}/{PU}_{DT}/{SM}_{PU}_{DT"}
  samplefmt: "{SM}/{SM}"
sample_project_run:
  runfmt: "{SM}/{PID}/{PU}_{DT}/{PID}_{PU}_{DT"}
  samplefmt: "{SM}/{SM}"
```

However, it is trivial to add more configurations, should that be deemed necessary.

ngs.settings

Warning: The ngs.settings section is slightly disorganized.

`ngs.settings` affect settings related to ngs analyses:

```
ngs.settings:
  annotation:
        annot_label: ""
        transcript_annot_gtf: ""
        sources: []
  db:
        dbsnp: ""
    ref: ref.fa
        transcripts: []
        build: ""
  fastq_suffix: ".fastq.gz"
  read1_label: "_1"
  read2_label: "_2"
  read1_suffix: ".fastq.gz"
  read2_suffix: ".fastq.gz"
  regions: []
  sequence_capture:
    bait_regions: []
        target_regions: []
```

For all settings, see `rules/ngs.settings`.

samples

The `samples` section is one of the few top-level configuration keys that are actually set, in this case to a list of sample names.

Application settings

Applications, i.e. bioinformatics software, are grouped in sections by their application name. Subsections correspond to rules, or subprograms. For instance, the entire bwa section looks as follows (with a slight abuse of notation as we here mix yaml with python objects):

```
bwa:
  cmd: bwa
  ref: config['ngs.settings']['db']['ref']
  index: ""
  index_ext: ['.amb', '.ann', '.bwt', '.pac', '.sa']
  threads: config['settings']['threads']
  mem:
    options:
```

Setting option `threads` would then override the value in `settings`, providing a means to fine-tune options on a per-application basis.

Workflow settings

Finally, the workflows comes with a configuration section called `workflow`.

2.2 Nextflow

TODO.

CHAPTER 3

Developer guide

3.1 Summary

- use the issue tracker
- create feature branches and submit pull requests

3.2 Setting up a local copy

See installation section *From sources*.

3.3 Branching/development model

The development model is based on a stable *master* branch and an unstable *develop* branch. The *master* branch should be used in production. Pushing to master/develop has been disabled; only pull requests from feature branches are permitted.

Vincent Driessen's branching model provides a good model for organizing the development process, and its adoption is recommended here. It adds some more branch types, of which three will be descriped below: feature, release and hotfix branches.

3.3.1 Feature branches

Feature branches hold new features for upcoming releases and are branched off develop. Feature branches should be prefixed with "feature/":

```
$ git checkout develop
$ git checkout -b "feature/myfeature"
```

Once you've added the feature you want, push the branch to bitbucket and create a pull request to the develop branch.

3.3.2 Release branches

Release branches group recent feature changes into a release set. More code additions can be made, although they should focus on minor fixes. Documentation updates are encouraged.

When creating a release branch, make sure you branch off develop. Furthermore, release branches should be prefixed with "release/":

```
$ git checkout develop
$ git checkout -b "release/X.X.X"
```

Upon creating a release branch, the version number must be bumped, preferably with *bumpversion*. First check that the changes do what you expect:

```
$ bumpversion --dry-run --verbose --new-version X.X.X part
```

Note that the part argument should be set to either patch, minor or major, depending on what part is to be updated. If everything looks ok, bump the version with

```
$ bumpversion --new-version X.X.X part
```

The configuration is setup not to tag the commit. Since merging is done on bitbucket, the release tag must be assigned manually to the master branch, once the release been merged.

When the release branch is finished it is merged into master and master is backmerged into develop. See the gitflow cheat sheet for more information.

3.3.3 Hotfixes

Occasionally, things break on the master branch that require immediate fixing. This is what hotfixes are for. Importantly, updates to master must also be merged immediately with develop to keep it in sync. gitflow (see following section) has builtin support for hotfixes.

Ideally, *all* hotfixes should be accompanied by a regression test, so that the error doesn't pop up again.

3.3.4 Using gitflow

It may help to use gitflow to organise work, as it is based on Vincent Driessen's branching model. The commands simplify the task of creating feature branches and hotfixes.

To setup gitflow, issue the following in the source code directory:

```
$ git flow init
```

which produces (press ENTER at each question):

```
Which branch should be used for bringing forth production releases?
   - master
Branch name for production releases: [master]
Branch name for "next release" development: [develop]

How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
```

```
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [/path/to/source/.git/hooks]
```

Then, to create a feature branch, simply type

```
$ git flow feature start test
```

which produces

```
Switched to a new branch 'feature/test'

Summary of actions:
  - A new branch 'feature/test' was created, based on 'develop'
  - You are now on branch 'feature/test'

Now, start committing on your feature. When done, use:

    git flow feature finish test
```

Warning: do not issue the finish command locally as it will merge the feature branch into develop. Merging is only done on bitbucket.

3.4 Issues

For all problems, small or large, use the issue tracker instead of sending emails! The main motivation is that all developers should be able to follow the discussion and history of any issue of general interest.

3.5 Adding a workflow

Note: WIP: Describe minimum requirements, including

1. tests for all sample organizations
2. example snakefiles and configurations

3.6 Continuous integration

As the number of collaborators on a project grows, code integration problems frequently occur. Continuous integration is a method for dealing with these issues. Typically, whenever a push is done to the repository, tests are automatically run on a test server. bitbucket has recently added a service called *Pipelines* which gives some support for CI. It runs integration tests in Docker containers. `lts-workflows` provides a Docker container that packages the basic dependencies for running tests. Each workflow then provides a separate Docker container that builds on the lts-workflows container, adding workflow-specific dependencies, for running tests.

CHAPTER 4

The workflow test environment

The workflow test environment[1] is built using the pytest framework. All workflows have tests that test minimal features of a workflow, such as listing rules or printing help messages. In addition, by installing the pytest plugin pytest_ngsfixtures, the workflows can be tested on small data sets.

4.1 Running workflow tests

There are two different ways to run the tests, depending on installation mode. If the _workflow was installed as a package, either via `python setup.py install` or a `conda` install, running

```
$ pytest --pyargs workflow
```

will run the tests. In addition, if pytest_ngsfixtures is installed, the workflow will be run on a small test data set.

`lts_workflows` provides a number of pytest options (see *Additional options*). Unfortunately, they are not loaded when running the tests as described above. Rather, the full path to the test file must be given for the options to load:

```
$ pytest /path/to/workflow/tests/ -h
```

The test suite will first setup and install local conda environments necessary for the tests, and then run the tests. Please note that the intended use of the local conda environments is to run the tests only, **not** to run analyses based on the workflows.

Alternatively, by applying the `-D` option the test conda enviroment setup is disabled. This obviously requires that the dependencies are already installed (see section Installing dependencies below).

4.2 Additional options

`lts_workflows` provides a helper function `lts_workflows.pytest.plugin.addoptions()` for adding

[1] This section does **not** describe how to run the test suite for `lts-workflows`. Rather, it describes general features of running workflow tests. Obviously, a workflow has to be installed for this section to apply.

pytest options. Depending on the workflow engine, different options are added. As an example, running the following setup code

```
def pytest_addoption(parser):
    group = parser.getgroup("ltssm_scrnaseq", "single cell rna sequencing options")
    lts_pytest.addoption(group)
```

in a pytest conftest file will add the following options to the test suite:

```
single cell rna sequencing options:
  --no-slow             don't run slow tests
  -H, --hide-workflow-output
                        hide workflow output
  -T THREADS, --threads=THREADS
                        number of threads to use
  -D, --disable-test-conda
                        disable test conda setup; instead use user-supplied
                        environments, where the activated environment hosts
                        snakemake
  --conda-install-dir=CONDA_INSTALL_DIR
                        set conda install dir
  --conda-update        update local conda installation
  -2 PYTHON2_CONDA, --python2-conda=PYTHON2_CONDA
                        name of python2 conda environment [default: py2.7]
  -C, --use-conda       pass --use-conda flag to snakemake workflows; will
                        install conda environments on a rule by rule basis
```

All tests that execute workflows have been marked as *slow*. To disable these tests, add the `--no-slow` option. By default, workflow output is sent to stdout which is captured. If you want to follow progress, add the regular pytest `-s` option. The `-T` option states how many threads/processes snakemake will use and can be set to increase the speed of the slow tests. Finally, the test environment will check if there is a conda environment called `py2.7` and if so, add the bin path to `PATH`. Use the `-2` option if your python2 conda environment is named differently.

Note that the workflow directories should contain conda environment files `environment.yaml` and `environment-27.yaml` that define the depencies for a workflow. You can apply the latter to you python2 repository by issuing

```
$ conda env update -n python2env -f environment-27.yaml
```

4.3 Local conda installs

By default, the test setup will automatically download and install all required packages via conda to `$HOME/.conda_env`. By passing the option `--disable-test-conda` (or `-D`), dependencies will not be installed by default. The following sections describe the steps needed to setup personal conda environments with the required packages.

4.3.1 Installing dependencies with deploy_workflow.py

> **Warning:** Due to refactorization, this is currently broken; see issue #1.

The helper script `deploy_workflow.py` can be employed to install required workflow dependencies in user-specified conda environments.

4.3.2 Semi-automated installation of snakemake and dependencies

Setup a conda python3 environment that hosts snakemake:

```
$ conda create -n py3.5 -c bioconda snakemake python=3.5
```

Some workflows have python2 program dependencies. Create a conda environment for these packages too:

```
$ conda create -n py2.7  python=2.7
```

Every workflow has a conda environment file, `environment.yaml` and possibly `environment-27.yaml` that list the necessary dependencies. You can update your conda python environments like so:

```
$ conda env update -n=py3.5 -f /path/to/environment.yaml
$ conda env update -n=py2.7 -f /path/to/environment-27.yaml
```

4.3.3 Semi-automated installation of snakemake and dependencies

Unfortunately, nextflow requires java sdk <=8.0, whereas gatk requires java sdk >=8.0. For this reason, it is recommended to install nextflow in a separate conda environment:

```
$ conda create -n py3.5 -c bioconda nextflow python=3.5
```

4.4 Test fixtures

TODO.

4.5 Hints on developing workflows

Use the test run wrapper functions in `lts_workflow.pytest.helpers` to setup tests. They will create a file `command.sh` located in the test output directory that can be rerun to aid in debugging.

4.6 Testing external data sources

If you have data that you want to test, bot whose sample layout is not yet provided by the fixtures, you have to run snakemake as usual:

```
$ snakemake -s /path/to/Snakefile -d /path/to/sample_data --configfile /path/to/
↪config.yaml targetname
```

You then obviously need to create a config file and a sampleinfo file. You can also use the factory functions in pytest_ngsfixtures to generate custom fixtures that resemble your sample layout.

CHAPTER 5

Docker images

`lts-workflows` includes a docker image *percyfal/lts-workflows* that serves as a base image for workflows in the `lts-workflows` package. It is configured to be fairly lean, containing packages for reproducible research and literate programming using R and Rmarkdown. Workflows that provide docker images should use *percyfal/lts-workflows* as the starting image.

CHAPTER 6

Workflows

Currently the following workflows are available:

- lts-workflows-sm-non-model-toolkit: Snakemake toolkit for analysis of non-model organisms, including workflows and rules for doing variant calling, BQSR, VQSR, and demographic modelling
- lts-workflows-sm-scrnaseq: Snakemake workflow for single-cell RNA seqnames

CHAPTER 7

Credits

7.1 Development Lead

- Per Unneberg <per.unneberg at scilifelab.se>
- Rasmus Ågren <rasmus.agren at scilifelab.se>
- Leif Väremo Wigge <leif.varemo at scilifelab.se>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

8.1 0.2.5 (2018-04-05)

- Change to use Ubuntu as base Docker image and removed LaTeX packages
- Removed Nextflow environment

8.2 0.2.4 (2017-11-14)

- Hotfix: remove versioneer from setup_requirements (issue #24)

8.3 0.2.3 (2017-11-14)

- Add setup requirements to install tagged lts-workflows version in docker image

8.4 0.2.2 (2017-11-13)

- Add options to snakemake_run (issue #23)

8.5 0.2.1 (2017-09-26)

- Minor changes to conda/meta.yaml
- Update docs
- Update development requirements
- Add snakemake utilities

- Add inconsolata fonts to docker
- CRLF to LF and Dockerfile organization
- Make conda builds with conda build-all (issue #22)
- Fix pytest mark for slow tests (issue #19)
- Add pytest entry point (issue #18)
- Add configfile option to pytest (issue #16)
- Redirect subprocess stderr to stdout and use stdout variable (issue #17)

8.6 0.2.0 (2017-03-21)

- Add docker base image and make it smallish (issue #3)
- Update docs

8.7 0.1.1 (2017-03-01)

- Convert threads argument to string (issue #7)
- Add population layouts to helper function (issue #4)

8.8 0.1.0 (2017-02-12)

- First release on conda.

CHAPTER 9

lts_workflows

9.1 lts_workflows package

9.1.1 Subpackages

lts_workflows.pytest package

Submodules

lts_workflows.pytest.factories module

lts_workflows.pytest.helpers module

lts_workflows.pytest.plugin module

Module contents

lts_workflows.snakemake package

Submodules

lts_workflows.snakemake.config module

Module contents

9.1.2 Submodules

9.1.3 lts_workflows.utils module

9.1.4 Module contents

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

l

Index

L