Popper Documentation

Release 2.x

Ivo Jimenez

Sep 29, 2021

Contents

CHAPTER 1

Getting Started

Before going through this guide, you need to have the Docker engine installed on your machine (see installations instructions here). In addition, this guide assumes familiarity with Linux containers and the container-native paradigm to software development. You can read a high-level introduction to these concepts in this page, where you can also find references to external resources that explain them in depth.

1.1 Installation

To install or upgrade Popper, run the following in your terminal:

```
curl -sSfL https://raw.githubusercontent.com/getpopper/popper/master/install.sh | sh
```

1.2 Create Your First Workflow

Assume that as part of our work we want to carryout two tasks:

1. Download a dataset (CSV) that we know is available at https://github.com/datasets/co2-fossil-global/raw/master/global.csv
2. Modify the dataset, specifically we want to get the transpose of the this CSV table.

For the first task we can use `curl`, while for the second we can use `csvtool`.

When we work under the container-native paradigm, instead of going ahead and installing these on our computer, we first look for available images on a container registry, for example https://hub.docker.com, to see if the software we need is available.

In this case we find two images that do what we need and proceed to write this workflow in a `wf.yml` file using your favorite editor:

```
steps:
# download CSV file with data on global CO2 emissions
- id: download
  uses: docker://byrnedo/alpine-curl:0.1.8
  args: [-LO, https://github.com/datasets/co2-fossil-global/raw/master/global.csv]

# obtain the transpose of the global CO2 emissions table
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -o, global_transposed.csv]
```

1.3 Run your workflow

To execute the workflow you just created:

```
popper run -f wf.yml
```

Since this workflow consists of two steps, there were two corresponding containers that were executed by the underlying container engine, which is Docker in this case. We can verify this by asking Docker to show the list of existing containers:

```
docker ps -a
```

You should see the two containers from the example workflow being listed along with other containers. The name of the containers created by popper are prefixed with `popper_`. To obtain more detailed information of what the `popper run` command does, you can pass the `--help` flag to it:

```
popper run --help
```

> **TIP**: All popper subcommands allow you to pass `--help` flag to it to get more information about what the command does.

1.4 Debug your workflow

From time to time, we find ourselves with a step that does not quite do what we want it to. In these cases, we can open an interactive shell instead of having to update the YAML file and invoke `popper run` again. In those cases, the `popper sh` comes handy. For example, if we would like to explore what other things can be done inside the container for the second step:

```
popper sh -f wf.yml get-transpose
```

And the above opens a shell inside a container instantiated from the `docker.io/getpopper/csvtool:2.4` image. In this shell we can, for example, obtain information about what else can the `csvtool` do:

```
csvtool --help
```

Based on this exploration, we can see that we can pass a `-u TAB` flag to the `csvtool` in order to generate a tab-separated output file instead of a comma-separated one. Assuming this is what we wanted to achieve in our case, we then quit the container by running `exit`.

Back on our host machine context, that is, not running inside the container anymore, we can update the second step by editing the YAML file to look like the following:

```
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -u, TAB, -o, global_transposed.csv]
```

And test that what we changed worked by running in non-interactive mode again:

```
popper run -f wf.yml get-transpose
```

1.5 Next Steps

- Learn more about all the CLI features that Popper provides.
- Take a look at the "Workflow Language" for the details on what else can you specify as part of a Step's attributes.
- Read the "Popper Execution Runtime" section to learn more about what other execution environments Popper supports, as well as how to customize the behavior of the underlying execution.
- Browse existing workflow examples.
- Take a self-paced tutorial to learn how to use other features of Popper.

CHAPTER 2

CLI features

2.1 New workflow initialization

Create a Git repository:

```
mkdir mypaper
cd mypaper
git init
echo '# mypaper' > README.md
git add .
git commit -m 'first commit'
```

Initialize the popper repository and add the configuration file to git:

```
popper init
git add .
git commit -m 'adds .popper.yml file'
```

Initialize a workflow

```
popper scaffold
```

Show what this did (a `wf.yml` should have been created):

```
ls -l
```

Commit the "empty" pipeline:

```
git add .
git commit -m 'adding my first workflow'
```

2.2 Executing a workflow

To run the workflow:

```
popper run -f wf.yml
```

where `wf.yml` is a file containing a workflow.

2.3 Executing a step interactively

For debugging a workflow, it is sometimes useful to open a shell inside a container associated to a step of a workflow. To accomplish this, run:

```
popper sh <STEP>
```

where `<STEP>` is the name of a step contained in the workflow. For example, given the following workflow:

```
steps:
- id: mystep
  uses: docker://ubuntu:18.04
  runs: ["ls", "-l"]
  dir: /tmp/
  env:
    MYENVVAR: "foo"
```

if we want to open a shell that puts us inside the `mystep` above (inside an container instance of the `ubuntu:18.04` image), we run:

```
popper sh mystep
```

And this opens an interactive shell inside that step, where the environment variable `MYENVVAR` is available. Note that the `runs` and `args` attributes are overridden by Popper. By default, `/bin/bash` is used to start the shell, but this can be modified with the `--entrypoint` flag.

2.4 Parametrizing workflows with substitutions

A workflow can be parametrized by making use of substitutions. A substitution is a string in the YAML file with the `$_` prefix, for example:

```
steps:
- id: mystep
  uses: docker://alpine:$_ALPINE_VERSION
  runs: ["ls", "-l"]
```

in the above workflow, the `$_ALPINE_VERSION` string defines a substitution, and will be replaced by a value defined in the command line via the `--substutition` or `-s` flags:

```
popper run -s _ALPINE_VERSION=3.12 -f wf.yml
```

2.5 Customizing container engine behavior

By default, Popper instantiates containers in the underlying engine by using basic configuration options. When these options are not suitable to your needs, you can modify or extend them by providing engine-specific options. These options allow you to specify fine-grained capabilities, bind-mounting additional folders, etc. In order to do this, you can provide a configuration file to modify the underlying container engine configuration used to spawn containers. This is a YAML file that defines an `engine` dictionary with custom options and is passed to the `popper run` command via the `--conf` (or `-c`) flag.

For example, to make Popper spawn Docker containers in privileged mode, we can write the following option:

```
engine:
    name: docker
    options:
       privileged: True
```

Similarly, to bind-mount additional folders, we can use the `volumes` option to list the directories to mount:

```
engine:
    name: docker
    options:
       privileged: True
       volumes:
       - myvol1:/folder
       - myvol2:/app
```

Assuming the above is stored in a file called `config.yml`, we pass it to Popper by running:

```
popper run -f wf.yml -c config.yml
```

NOTE:

Currently, the `--conf` option is only supported for the `docker`engine.

2.6 Continuously validating a workflow

The `ci` subcommand generates configuration files for multiple CI systems. The syntax of this command is the following:

```
popper ci --file wf.yml <service-name>
```

Where `<name>` is the name of the CI system (see `popper ci --help` to get a list of supported systems). If the `wf.yml` workflow makes use of substitutions, we can create a matrix by doing:

```
popper ci -f wf.yml travis -s _P1=p1v1 -s _P1=p1v2 -s _P2=p2v1 -s _P2=p2v2
```

And the above will create a 2x2 matrix job, doing a parameter sweep over the `_P1` and `_P2` given substitution values.

In the following, we show how to link github with some of the supported CI systems. In order to do so, we first need to create a repository on github and upload our commits:

```
# set the new remote
git remote add origin <your-github-repo-url>

# verify the remote URL
```

(continues on next page)

(continued from previous page)

```
git remote -v

# push changes in your local repository up to github
git push -u origin master
```

2.6.1 TravisCI

In the following, we assume we have an account on Travis CI. Assuming our repository is already on GitHub, we can enable it on TravisCI so that it is continuously validated (see here for a guide). Once the project is registered on Travis, we proceed to generate a `.travis.yml` file:

```
cd my-repo/

popper ci --file wf.yml travis
```

Before we can execute tests on travis, we need to commit the file we just generated:

```
git add .travis.yml
git commit -m 'Adds TravisCI config file'
```

We then can trigger an execution by pushing to GitHub:

```
git push
```

After this, one go to the TravisCI website to see your pipelines being executed. Every new change committed to a public repository will trigger an execution of your pipelines. To avoid triggering an execution for a commit, include a line with `[skip ci]` as part of the commit message.

> **NOTE**: TravisCI has a limit of 2 hours, after which the test is terminated and failed.

Job Matrix

If the workflow is parametrized by the use of substitutions, we can create a matrix. For example, assume a workflow like the following:

```
steps:
- id: mystep
  uses: docker://alpine:$_ALPINE_VERSION
  runs: [sh, -cue]
  args:
  - |
    # execute command with parameter
    ls -l $_FOLDER
```

```
popper ci travis \
  -f wf.yml \
  -s _ALPINE_VERSION=3.10 \
  -s _ALPINE_VERSION=3.11 \
  -s _ALPINE_VERSION=3.12 \
  -s _FOLDER=/root \
  -s _FOLDER=/etc \
  -s _FOLDER=/usr
```

And the above will create a 3x3 matrix job for travis.

2.7 Translating workflows for other task runners

The `translate` subcommand generates configuration files for other CI systems. The file generated by the `ci` subcommand executes Popper internally, the `translate` subcommand convert Popper workflows directly to the notation of the target CI system. The syntax of this command is the following:

```
popper translate --file wf.yml --to <service-name>
```

Where `<service-name>` is the name of the CI system (see `popper translate --help` to get a list of supported systems).

2.7.1 Drone

The `translate` subcommand supports Drone CI. The command converts a Popper workflow to a Drone pipeline.

Restrictions on translation are as follows:

- Running commands on Docker and Host machine is supported. Singularity and Podman are not supported.
- All steps in a workflow must use either Docker or the host machine. The two cannot be combined in a single workflow.
- Only pre-built Docker images can be used. Workflows that specify the directory where the Dockerfile is located in the `uses` attribute cannot be translated.
- If you specify the `dir` attribute, you must also specify the `runs` attribute.

2.7.2 Task

The `translate` subcommand supports Task. The command converts a Popper workflow to a Taskfile.

Restrictions on translation are as follows:

- Running commands on Docker and Host machine is supported. Singularity and Podman are not supported.
- Only pre-built Docker images can be used. Workflows that specify the directory where the Dockerfile is located in the `uses` attribute cannot be translated.
- The Popper workflow to translate must not have a step with an ID of `default`

2.8 Visualizing workflows

While `.workflow` files are relatively simple to read, it is nice to have a way of quickly visualizing the steps contained in a workflow. Popper provides the option of generating a graph for a workflow. To generate a graph for a pipeline, execute the following:

```
popper dot -f wf.yml
```

The above generates a graph in `.dot` format. To visualize it, you can install the `graphviz` package and execute:

```
popper dot -f wf.yml | dot -T png -o wf.png
```

The above generates a `wf.png` file depicting the workflow. Alternatively you can use the http://www.webgraphviz.com/ website to generate a graph by copy-pasting the output of the `popper dot` command.

CHAPTER 3

Concepts

The main three concepts behind Popper are Linux containers, the container-native paradigm, and workflows. This page is under construction, we plan on expanding it with our own content (contributions are more than welcome)! For now, we provide with a list of external resources and a Glossary.

3.1 Resources

Container Concepts:

- Overview of Containers in Red Hat Systems (Red Hat)
- An Introduction to Containers (Rancher)
- A Beginner-Friendly Introduction to Containers, VMs and Docker (freecodecamp.org)
- A Practical Introduction to Container Terminology (Red Hat)

Container-native paradigm:

- 5 Reasons You Should Be Doing Container-native Development (Microsoft)
- Let's Define "Container-native" (TechCrunch)
- The 7 Characteristics of Container-native Infrastructure (Joyent)

Docker:

- A Docker tutorial for beginners
- Dockerfile tutorial by example

Singularity:

- Introduction to Singularity

3.2 Glossary

- **Linux containers**. An OS-level virtualization technology for isolating applications in a Linux host machine.
- **Container runtime**. The software that interacts with the Linux kernel in order to provide with container primitives to upper-level components such as a container engine (see "Container Engine"). Examples of runtimes are runc, Kata and crun.
- **Container engine**. Container management software that provides users with an interface to. Examples of engines are Docker, Podman and Singularity.
- **Container-native development**. An approach to writing software that makes use of containers at every stage of the software delivery cycle (building, testing, deploying, etc.). In practical terms, when following a container-native paradigm, other than a text editor or ID, dependencies required to develop, test or deploy software are NEVER installed directly on your host computer. Instead, they are packaged in container images and you make use of them through a container engine.
- **Workflow**. A series of steps, where each step specifies what it does, as well as which other steps need to be executed prior to its execution. It is commonly represented as a directed acyclic graph (DAG), where each node represents a step. The word "pipeline" is usually used interchangeably to refer to a workflow.
- **Task or Step**. A node in a workflow DAG.
- **Container-native workflow**. A workflow where each step runs in a container.
- **Container-native task or step**. A step in a container-native workflow that specifies the image it runs, the arguments that are executed, the environment available inside the container, among other attributes available for containers (network configuration, resource limits, capabilities, volumes, etc.).

CHAPTER 4

Workflow Syntax and Execution Runtime

This section introduces the YAML syntax used by Popper, describes the workflow execution runtime and shows how to execute workflows in alternative container engines.

4.1 Syntax

A Popper workflow file looks like the following:

```
steps:
- uses: docker://alpine:3.9
  args: ["ls", "-la"]

- uses: docker://alpine:3.11
  args: ["echo", "second step"]

options:
  env:
    FOO: BAR
  secrets:
  - TOP_SECRET
```

A workflow specification contains one or more steps in the form of a YAML list named `steps`. Each item in the list is a dictionary containing at least a `uses` attribute, which determines the docker image being used for that step. An `options` dictionary specifies options that are applied to the workflow.

4.1.1 Workflow steps

The following table describes the attributes that can be used for a step. All attributes are optional with the exception of the `uses` attribute.

4.1.2 Referencing images in a step

A step in a workflow can reference a container image defined in a `Dockerfile` that is part of the same repository where the workflow file resides. In addition, it can also reference a `Dockerfile` contained in public Git repository. A third option is to directly reference an image published a in a container registry such as DockerHub. Here are some examples of how you can refer to an image on a public Git repository or Docker container registry:

It's strongly recommended to include the version of the image you are using by specifying a SHA or Docker tag. If you don't specify a version and the image owner publishes an update, it may break your workflows or have unexpected behavior.

In general, any Docker image can be used in a Popper workflow, but keep in mind the following:

- When the `runs` attribute for a step is used, the `ENTRYPOINT` of the image is overridden.
- The `WORKDIR` is overridden and `/workspace` is used instead (see *The Workspace* section below).
- The `ARG` instruction is not supported, thus building an image from a `Dockerfile` (public or local) only uses its default value.
- While it is possible to run containers that specify `USER` other than root, doing so might cause unexpected behavior.

4.1.3 Referencing private Github repositories

You can reference Dockerfiles located in private Github repositories by defining a `GITHUB_API_TOKEN` environment variable that the `popper run` command reads and uses to clone private repositories. The repository referenced in the `uses` attribute is assumed to be private and, to access it, an API token from Github is needed (see instructions here). The token needs to have permissions to read the private repository in question. To run a workflow that references private repositories:

```
export GITHUB_API_TOKEN=access_token_here
popper run -f wf.yml
```

If the access token doesn't have permissions to access private repositories, the `popper run` command will fail.

4.1.4 Workflow options

The `options` attribute can be used to specify `env` and `secrets` that are available to all the steps in the workflow. For example:

```
options:
  env:
    FOO: var1
    BAR: var2
  secrets: [SECRET1, SECRET2]

steps:
- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $FOO $SECRET1"]

- uses: docker://alpine:3.11
  runs: sh
  args: ["-c", "echo $ONLY_FOR"]
  env:
    ONLY_FOR: this step
```

The above shows environment variables that are available to all steps that get defined in the `options` dictionary; it also shows an example of a variable that is available only to a single step (second step). This attribute is optional.

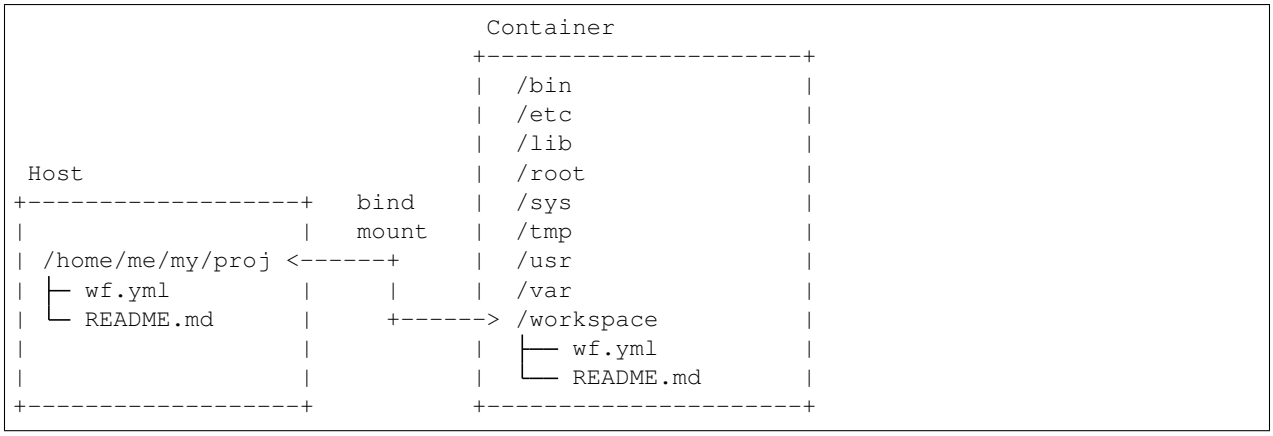
4.2 Execution Runtime

This section describes the runtime environment where a workflow executes.

4.2.1 The Workspace

When a step is executed, a folder in your machine is bind-mounted (shared) to the `/workspace` folder inside the associated container. By default, the folder being bind-mounted is `$PWD`, that is, the working directory from where `popper run` is being invoked from. If the `-w` (or `--workspace`) flag is given, then the value for this flag is used instead. See the official Docker documentation for more information about how volumes work with containers.

The following diagram illustrates this relationship between the filesystem namespace of the host (the machine where `popper run` is executing) and the filesystem namespace within container:



For example, let's look at a workflow that creates files in the workspace:

```
steps:
- uses: docker://alpine:3.12
  args: [touch, ./myfile]
```

The above workflow has only one single step that creates the `myfile` file in the workspace directory if it doesn't exist, or updates its metadata if it already exists, using the `touch` command. Assuming the above workflow is stored in a `wf.yml` file in `/home/me/my/proj/`, we can run it by first changing the current working directory to this folder:

```
cd /home/me/my/proj/
popper run -f wf.yml
```

And this will result in having a new file in `/home/me/my/proj/myfile`. However, if we invoke the workflow from a different folder, the folder being bind-mounted inside the container is a different one. For example:

```
cd /home/me/
popper run -f /home/me/my/proj/wf.yml
```

In the above, the file will be written to `/home/me/myfile`, because we are invoking the command from `/home/me/`, and this path is treated as the workspace folder. If we provide a value for the `--workspace` flag (or its short version `-w`), the workspace path then changes and thus the file is written to this given location. For example:

```
cd /
popper run -f /home/me/my/proj/wf.yml -w /home/me/my/proj/
```

The above writes the `/home/me/my/proj/myfile` even though Popper is being invoked from `/`. Note that the above is equivalent to the first example of this subsection, where we first changed the directory to `/home/me/my/proj` and ran `popper run -f wf.yml`.

4.2.2 Changing the working directory

To specify a working directory for a step, you can use the `dir` attribute in the workflow, which takes as value a string representing an absolute path inside the container. This changes where the specified command is executed. For example, adding `dir` as follows:

```
steps:
- uses: docker://alpine:3.9
  args: [touch, ./myfile]
  dir: /tmp/
```

And assuming that it is stored in `/home/me/my/proj/wf.yml`, invoking the workflow as:

```
cd /home/me
popper run -f wf.yml -w /home/me/my/proj
```

Would result in writing `myfile` in the `/tmp` folder that is **inside** the container filesystem namespace, as opposed to writing it to `/home/me/my/projc/` (the value given for the `--workspace` flag). As it is evident in this example, if the directory specified in the `dir` attribute resides outside the `/workspace` folder, then anything that gets written to it won't persist after the step ends its execution (see "Filesystem namespaces and persistence" below for more).

For completeness, we show an example of using `dir` to specify a folder within the workspace:

```
steps:
- uses: docker://alpine:3.9
  args: [touch, ./myfile]
  dir: /workspace/my/proj/
```

And executing:

```
cd /home/me
popper run -f wf.yml
```

would result in having a file in `/home/me/my/proj/myfile`.

4.2.3 Filesystem namespaces and persistence

As mentioned previously, for every step Popper bind-mounts (shares) a folder from the host (the workspace) into the `/workspace` folder in the container. Anything written to this folder persists. Conversely, anything that is NOT written in this folder will not persist after the workflow finishes, and the associated containers get destroyed.

4.2.4 Environment variables

A step can define, read, and modify environment variables. A step defines environment variables using the `env` attribute. For example, you could set the variables `FIRST`, `MIDDLE`, and `LAST` using this:

```
steps:
- uses: "docker://alpine:3.9"
  args: ["sh", "-c", "echo my name is: $FIRST $MIDDLE $LAST"]
  env:
    FIRST: "Jane"
    MIDDLE: "Charlotte"
    LAST: "Doe"
```

When the above step executes, Popper makes these variables available to the container and thus the above prints to the terminal:

```
my name is: Jane Charlotte Doe
```

Note that these variables are only visible to the step defining them and any modifications made by the code executed within the step are not persisted between steps (i.e. other steps do not see these modifications).

Git Variables

When Popper executes insides a git repository, it obtains information related to Git. These variables are prefixed with `GIT_` (e.g. to `GIT_COMMIT` or `GIT_BRANCH`).

4.2.5 Exit codes and statuses

Exit codes are used to communicate about a step's status. Popper uses the exit code to set the workflow execution status, which can be `success`, `neutral`, or `failure`:

4.3 Container Engines

By default, Popper workflows run in Docker on the machine where `popper run` is being executed (i.e. the host machine). This section describes how to execute in other container engines. See *next section* for information on how to run workflows on resource managers such as SLURM and Kubernetes.

To run workflows on other container engines, an `--engine <engine>` flag for the `popper run` command can be given, where `<engine>` is one of the supported ones. When no value for this flag is given, Popper executes workflows in Docker. Below we briefly describe each container engine supported, and lastly describe how to pass engine-specific configuration options via the `--conf` flag.

4.3.1 Docker

Docker is the default engine used by the `popper run`. All the container configuration for the docker engine is supported by Popper. Popper also supports running workflows on remote docker daemons by use of the `DOCKER_HOST`, `DOCKER_TLS_VERIFY` and `DOCKER_CERT_PATH` variables, as explained in the official documentation. For example:

```
export DOCKER_HOST="ssh://myuser@hostname"
popper run -f wf.yml
```

The above runs the workflow on the `hostname` machine instead of locally. It assumes the following:

1. `myuser` has passwordless access to `hostname`, otherwise the password to the machine is requested.
2. The `myuser` account can run `docker` on the remote machine.

4.3.2 Singularity

Popper can execute a workflow in systems where Singularity 3.2+ is available. To execute a workflow in Singularity containers:

```
popper run --engine singularity
```

Limitations

- The use of `ARG` in `Dockerfiles` is not supported by Singularity.
- The `--reuse` flag of the `popper run` command is not supported.

4.3.3 Host

There are situations when executing a command directly on the host where the `popper` command is running. This is done by making use of the special `sh` value for the `uses` attribute. This value instructs Popper to execute the command or script given in the `runs` attribute directly on the host. For example:

```
steps:
- uses: "sh"
  runs: ["ls", "-la"]

- uses: "sh"
  runs: "./path/to/my/script.sh"
  args: ["some", "args", "to", "the", "script"]
```

In the first step above, the `ls -la` command is executed on the workspace folder (see *"The Workspace"* section). The second one shows how to execute a script. Note that the command or script specified in the `runs` attribute are NOT executed in a shell. If you need a shell, you have to explicitly invoke one, for example:

```
steps:
- uses: sh
  runs: [bash, -c, 'sleep 10 && true && exit 0']
```

The obvious downside of running a step on the host is that, depending on the command being executed, the workflow might not be portable.

4.3.4 Custom engine configuration

Other than bind-mounting the `/workspace` folder, Popper runs containers with any default configuration provided by the underlying engine. However, a `--conf` flag is provided by the `popper run` command to specify custom options for the underlying engine in question (see here for more).

Alternatively, to restrict a configuration to a specific step in a workflow, set the desired parameters in the step's `options` **Note**: this is currently only supported for the Docker runtime

4.4 Resource Managers

By default, workflows are executed locally on the host where Popper is executed from. In addition, workflows can also be executed through other resource managers. The resource manager can be specified either through the

`--resource-manager/-r` option, or specified in the configuration file given via the `--config/-c` flag. If neither of them are provided, the steps are run in the host machine by default.

4.4.1 Kubernetes

Popper enables leveraging the compute and storage capabilities of the cloud by allowing running workflows on Kubernetes clusters. Users need to have access to a cluster config file in order to run workflows on Kubernetes. This file can be provided by a system administrator.

Popper provisions all the required resources and orchestrates the entire workflow execution. When a workflow is executed, Popper first creates a persistent volume claim, spawns an init pod and uses it to copy the workflow context (packed in the form of a `.tar.gz` file) into the persistent volume and then unpacks the context there. Subsequently, Popper tears down the init pod and executes the steps of a workflow in separate pods of their own. After the execution of each step, the respective pods are deleted but the persistent volume claim is not deleted so that it can be reused by subsequent workflow executions.

For running workflows on Kubernetes, several configuration options can be passed to the Kubernetes resource manager through the Popper configuration file to customize the execution environment. All the available configuration options have been described below:

- `namespace`: The namespace within which to provision resources like PVCs and Pods for workflow execution. If not provided the `default` namespace will be used.
- `persistent_volume_name`: Any pre-provisioned persistent volume like an NFS or EBS volume can be supplied through this option. Popper will then claim storage space from the supplied persistent volume. In the default case, a HostPath persistent volume of 1GB with a name of the form `pv-hostpath-popper-<workflowid>` will be created by Popper automatically.
- `volume_size`: The amount of storage space to claim from a persistent volume for use by a workflow. The default is 500MB.
- `pod_host_node`: The node on which to restrict the deployment of all the pods. This option is important when a HostPath persistent volume is used. In this case, users need to restrict all the pods to a particular node. If this option is not provided, Popper will leave the task of scheduling the pods upon Kubernetes. The exception to this is, when both the `pod_host_node` and `persistent_volume_name` options are not provided, Popper will try to find out a pod and schedule all the pods (init-pods + step-pods) on that node to use the `HostPath` persistent volume of 1GB which will be automatically created.
- `hostpathvol_path`: The path to use for creating a HostPath volume. If not provided, /tmp will be used.
- `hostpathvol_size`: The size of the HostPath volume. If not provided, 1GB will be used.

To run workflows on Kubernetes:

```
$ popper run -f wf.yml -r kubernetes
```

Limitations

- A workflow cannot build local Dockerfiles. In order to work around this issue, a workflow can build an image using BuildKit or Kaniko as explained here.

4.4.2 SLURM

Popper workflows can run on HPC (Multi-Node environments) using Slurm as the underlying resource manager to distribute the execution of a step to several nodes. You can get started with running Popper workflows through Slurm by following the example below.

NOTE: Set the `POPPER_CACHE_DIR` environment variable to `/path/to/shared/.cache` while running a workflow on multiple nodes.

Let's consider a workflow `sample.yml` like the one shown below.

```
steps:
- id: one
  uses: docker://alpine:3.9
  args: ["echo", "hello-world"]

- id: two
  uses: popperized/bin/sh@master
  args: ["ls", "-l"]
```

To run all the steps of the workflow through SLURM resource manager, use the `--resource-manager` or `-r` option of the `popper run` subcommand to specify the resource manager.

```
popper run -f sample.yml -r slurm
```

This runs the workflow on a single compute node in the cluster which is also the default scenario when no specific configuration is provided.

To have more finer control on which steps to run through SLURM resource manager, the specifications can be provided through the config file as shown below.

We create a config file called `config.yml` with the following contents.

```
engine:
  name: docker
  options:
    privileged: True
    hostname: example.local

resource_manager:
  name: slurm
  options:
    two:
      nodes: 2
```

Now, we execute `popper run` with this config file as follows:

```
popper run -f sample.yml -c config.yml
```

This runs the step `one` locally in the host and step `two` through SLURM on any 2 compute nodes. If `singularity` is used as the container engine, then by default the steps would run using MPI as SLURM jobs. This behaviour can be overriden by passing `mpi: false` in the configuration of the step for which MPI is not required.

4.5 Life of a Workflow

This section explains what popper does when it executes a workflow. We will break down what popper does behind the scenes when executing the following sample workflow, which can be found here:

```
steps:
# download CSV file with data on global CO2 emissions
- id: download
  uses: docker://byrnedo/alpine-curl:0.1.8
```

(continues on next page)

(continued from previous page)

```
  args: [-LO, https://github.com/datasets/co2-fossil-global/raw/master/global.csv]

# obtain the transpose of the global CO2 emissions table
- id: get-transpose
  uses: docker://getpopper/csvtool:2.4
  args: [transpose, global.csv, -o, global_transposed.csv]
```

Each step of a workflow has the following stages:

4.5.1 1. Look at `uses` attribute and pull/build image

Each step of a workflow must specify the `DockerFile` or Docker image it will use to create a container with a `uses` line. For example, the first step of our example workflow contains the following line:

```
uses: docker://byrnedo/alpine-curl:0.1.8
```

These statements may refer to a `Dockerfile` inside the same repository as the workflow; a `Dockerfile` inside an external, public repository or container registry; or an image in a registry.

The example `uses` line above would result in the following output from Popper:

```
[download] docker pull byrnedo/alpine-curl:0.1.8
```

This line indicates that the necessary image was successfully pulled by docker. If the image needs to be built from a Dockerfile, it will do so at this stage.

Popper would run this command under the hood if the engine used is Docker:

```
docker pull byrnedo/alpine-curl:0.1.8
```

and it would run this command if the engine is singularity:

```
singularity pull popper_download_f20ab8c9.sif docker://byrnedo/alpine-curl:0.1.8
```

The workings and limitations of `uses` and other possible attributes for a workflow are outlined here.

4.5.2 2. Configure and create container

Popper instantiates containers in the underlying engine (with Docker as the default) using basic configurations options. The underlying engine configuration can be modified using a configuration file. Learn more about configuring the engine here.

In the example workflow, the first step contains the following lines, one for the `id` (which is used as the name of the step) and one for the `args`:

```
id: download
```

```
args: [-LO, https://github.com/datasets/co2-fossil-global/raw/master/global.csv]
```

Using these inputs, Popper executes the following command for a Docker build:

```
docker create name=popper_download_f20ab8c9 byrnedo/alpine-curl:0.1.8 -LO https://
↪github.com/datasets/co2-fossil-global/raw/master/global.csv
```

This creates a docker container from the image given by the `uses` line with inputs from the `args` line, and with a name created using the id given in the `id` line and the id number of our specific workflow.

4.5.3 3. Launch container

Popper launches the container, waits for it to be done, and then prints the resulting output.

In the example workflow, the first step is run with the following commands when running in Docker and Singularity engines, respectively:

```
docker start
```

```
singularity run popper_download_f20ab8c9.sif (-LO, https://github.com/datasets/co2-
↪fossil-global/raw/master/global.csv)
```

This produces the following output:

```
[download] docker start
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   144    0   144    0     0    500      0 --:--:-- --:--:-- --:--:--   500
100  6453  100  6453    0     0  10509      0 --:--:-- --:--:-- --:--:-- 25709
Step 'download' ran successfully !
```

4.5.4 4. Move on to next step

The above three stages comprise a single step in a workflow's execution. As workflows can be made up of multiple steps, the workflow continues its execution by progressing to its next step, which contains its own `uses` and configurations for its containers and operations. Thus, your average workflow looks something like this:

```
steps:
- id: <optional step name>
  uses: <some local/public repository or container registry>
  args: [<command>, ..., <command>]

- id: <Optional step name>
  uses: <some local/public repository or container registry>
  args: [<command>, ..., <command>]
  .
  .
  .
```

The workflow repeats the same three stages for each step in the process. Consequently, the next step of our example workflow produces the following output:

```
[get-transpose] docker pull getpopper/csvtool:2.4
[get-transpose] docker create name=popper_get-transpose_f20ab8c9 image=getpopper/
↪csvtool:2.4 command=['transpose', 'global.csv', '-o', 'global_transposed.csv']
[get-transpose] docker start
Step 'get-transpose' ran successfully !
Workflow finished successfully.
```

Once the workflow has executed all of its outlined steps, its lifecycle is complete!

4.5.5 Conclusion

Hopefully this section has clarified how a Popper workflow iterates through its steps to simplify any workflow into a simple `popper run` call. Not only does it allow you to run fewer commands per run, it also runs the correct commands for different engines based on whether you're using Docker or Singularity.

Thus, Popper can be a useful tool for increasing efficiency on any workflow-heavy project!

CHAPTER 5

Guides

This is a list of guides related to several aspects of working with Popper workflows.

5.1 Choosing a location for your step

If you are developing a docker image for other people to use, we recommend keeping this image in its own repository instead of bundling it with your repository-specific logic. This allows you to version, track, and release this image just like any other software. Storing a docker image in its own repository makes it easier for others to discover, narrows the scope of the code base for developers fixing issues and extending the image, and decouples the image's versioning from the versioning of other application code.

5.2 Using shell scripts to define step logic

Shell scripts are a great way to write the code in steps. If you can write a step in under 100 lines of code and it doesn't require complex or multi-line command arguments, a shell script is a great tool for the job. When defining steps using a shell script, follow these guidelines:

- Use a POSIX-standard shell when possible. Use the `#!/bin/sh` shebang to use the system's default shell. By default, Ubuntu and Debian use the dash shell, and Alpine uses the ash shell. Using the default shell requires you to avoid using bash or shell-specific features in your script.
- Use `set -eu` in your shell script to avoid continuing when errors or undefined variables are present.

5.3 Hello world step example

You can create a new step by adding a `Dockerfile` to the directory in your repository that contains your step code. This example creates a simple step that writes arguments to standard output (`stdout`). An step declared in a `main.workflow` would pass the arguments that this step writes to `stdout`. To learn more about the instructions used in

the `Dockerfile`, check out the official Docker documentation. The two files you need to create an step are shown below:

./step/Dockerfile

```
FROM debian:9.5-slim

ADD entrypoint.sh /entrypoint.sh
ENTRYPOINT ["/entrypoint.sh"]
```

./step/entrypoint.sh

```
#!/bin/sh -l

sh -c "echo $*"
```

Your code must be executable. Make sure the `entrypoint.sh` file has `execute` permissions before using it in a workflow. You can modify the permission from your terminal using this command:

```
chmod +x entrypoint.sh
```

This `echos` the arguments you pass the step. For example, if you were to pass the arguments `"Hello World"`, you'd see this output in the command shell:

```
Hello World
```

5.4 Creating a Docker container

Check out the official Docker documentation.

5.5 Implementing a workflow for an existing set of scripts

This guide exemplifies how to define a Popper workflow for an existing set of scripts. Assume we have a project in a `myproject/` folder and a list of scripts within the `myproject/scripts/` folder, as shown below:

```
cd myproject/
ls -l scripts/

total 16
-rwxrwx---  1 user  staff   927B Jul 22 19:01 download-data.sh
-rwxrwx---  1 user  staff   827B Jul 22 19:01 get_mean_by_group.py
-rwxrwx---  1 user  staff   415B Jul 22 19:01 validate_output.py
```

A straight-forward workflow for wrapping the above is the following:

```
- uses: docker://alpine:3.12
  runs: "/bin/bash"
  args: ["scripts/download-data.sh"]

- uses: docker://alpine:3.12
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://alpine:3.12
```

(continues on next page)

(continued from previous page)

```
  args [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above runs every script within a Docker container. As you would expect, this workflow fails to run since the `alpine:3/12` image is a lightweight one (contains only Bash utilities), and the dependencies that the scripts need are not be available in this image. In cases like this, we need to either use an existing docker image that has all the dependencies we need, or create a docker image ourselves.

In this particular example, these scripts depend on CURL and Python. Thankfully, docker images for these already exist, so we can make use of them as follows:

```
- uses: docker://byrnedo/alpine-curl:0.1.8
  args: ["scripts/download-data.sh"]

- uses: docker://python:3.7
  args: ["./scripts/get_mean_by_group.py", "5"]

- uses: docker://python:3.7
  args: [
    "./scripts/validate_output.py",
    "./data/global_per_capita_mean.csv"
  ]
```

The above workflow runs correctly anywhere where Docker containers can run.

5.6 Building images using BuildKit

BuildKit can be used as part of a workflow to build a container image:

```
steps:
- id: build image using buildkit
  uses: docker://moby/buildkit:rootless
  runs: [buildctl-daemonless.sh]
  options:
    volumes:
    - $_DOCKER_CONFIG_DIR:/root/.docker/
  env:
    BUILDKITD_FLAGS: --oci-worker-no-process-sandbox
  args:
  - |
    build \
      --frontend dockerfile.v0 \
      --local context=/workspace/ \
      --local dockerfile=/workspace/my_container/Dockerfile \
      --import-cache type=registry,ref=docker.io/myrepo/myimg \
      --output type=image,name=docker.io/myrepo/myimg,push=true \
      --export-cache type=inline
```

The above uses BuildKit to build a container image from the `/workspace/my_container/Dockerfile` file and using `/workspace` as the build context. The `$_DOCKER_CONFIG_DIR` substitution is used to point to the directory where `buildctl` can find authentication credentials in order to pull the container images used as cache, as well as pushing the image produced by this step.

And the above workflow is executed by running:

```
popper run -f wf.yml -s _DOCKER_CONFIG_DIR=$HOME/.docker/
```

If credentials need to be generated as part of the execution of the workflow, the following step can be executed prior to running the BuildKit step:

```
- id: dockerhub login
  uses: docker://docker:19.03
  secrets: [DOCKERHUB_USERNAME, DOCKERHUB_PASSWORD]
  runs: [sh, -ec]
  options:
    volumes:
    - $_DOCKER_CONFIG_DIR:/root/.docker/
  args:
  - |
    docker login -u $DOCKERHUB_USERNAME -p $DOCKERHUB_PASSWORD
```

The above expects `DOCKERHUB_USERNAME` and `DOCKERHUB_PASSWORD` environment variables. Alternatively, these can be defined as substitutions:

```
- id: dockerhub login
  uses: docker://docker:19.03
  runs: [sh, -ec]
  options:
    volumes:
    - $_DOCKER_CONFIG_DIR:/root/.docker/
  args:
  - |
    docker login -u $_DOCKERHUB_USERNAME -p $_DOCKERHUB_PASSWORD
```

And executed as:

```
popper run -f wf.yml \
  -s _DOCKER_CONFIG_DIR=$PWD/docker-config/ \
  -s _DOCKERHUB_USERNAME=myuser \
  -s _DOCKERHUB_PASSWORD=mypass
```

5.7 Computational research with Python and JupyterLab

This guide explains how to use Popper to develop and run reproducible workflows for computational research in fields such as bioinformatics, machine learning, physics or statistics. Computational research with Python relies on complex software dependencies that are difficult to port across environments. In addition, a typical workflow involves multiple dependent steps which will be hard to replicate if not properly documented. Popper offers a solution to these challenges:

- Poppers abstracts over software environments with Linux containers.
- Poppers forces you to define your workflow explicetely such that it can be re-run in in a single command.

Popper thus provides an open-source alternative to managed solutions such as Code Ocean for reproducible computational research.

5.7.1 Pre-requisites

You should have basic knowledge of git, the command line and Python.

In addition, you should be familiar with the concepts introduced in the Getting Started section. This guide uses examples from machine learning but no prior knowledge of the field is required.

By default, this guide assumes that you use the Docker container engine, but highlights where the workflow will differ if you use another engine.

5.7.2 Getting started

The examples presented in this guide come from a workflow developed for the Flu Shot Learning research competition on Driven Data. This workflow shows examples of using Popper to automate common tasks in computational research:

- downloading data
- using a Jupyter notebook
- fitting/simulating a model
- visualizing the results
- generating a paper with up-to-date results

To help follow allong, see this repository with the final version of the workflow. To adapt the advice in this guide to your own project, get started with this Cookiecutter template for Popper.

Initial project structure:

```
├── LICENSE
├── README.md               <- The top-level README.
├── data                    <- The original, immutable data dump.
├── results
|   ├── models              <- Serialized models, predictions, model summaries.
|   └── figures             <- Graphics created during analysis.
├── paper                   <- Generated analysis as PDF, LaTeX.
|   ├── paper.tex
|   └── referenced.bib
└── src                     <- Python source code for this project.
    ├── notebooks           <- Jupyter notebooks.
    ├── get_data.sh         <- Script for downloading the original data dump.
    ├── models.py           <- Script defining models.
    ├── predict.py          <- Script for generating model predictions.
    └── evaluate_model.py   <- Script for generating model evaluation plots.
```

5.7.3 Getting data

Your workflow should automate downloading or generating data to ensure that it uses the correct, up-to-date version of the data. In this example, you can download data with a simple shell script:

```
#!/bin/sh
cd $1

wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/test_set_features.csv" -
↪-no-check-certificate
wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/training_set_labels.csv
↪" --no-check-certificate
wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/training_set_features.
↪csv" --no-check-certificate

echo "Files downloaded: $(ls)"
```

Now, wrap this step using a Popper workflow. In a new file `wf.yml` at the root of the folder,

```
steps:
- id: "dataset"
  uses: "docker://jacobcarlborg/docker-alpine-wget"
  args: ["src/get_data.sh", "data"]
```

Notes:

- pick a Docker image that contains the necessary utilities. For instance, a default Alpine image does not include `wget`.

5.7.4 Using JupyterLab

This sections explains how to use Popper to launch Jupyter notebooks, which are a useful tool for exploratory work. Refactoring successful experiments into your final workflow is easier if you keep the software environment consistent between both, which you can do by defining a container shared between steps.

Some workflows will require multiple containers (and `Dockerfiles`), so it is good practice to organize these from the start in a seperate folder. In `containers/`, create this `Dockerfile`:

```
FROM continuumio/miniconda3:4.8.2
ENV PYTHONDONTWRITEBYTECODE=true
# update conda environment with packages and clean up conda installation by removing
# conda cache/package tarbarlls and python bytecode
COPY containers/environment.yml .
RUN conda env update -f environment.yml \
    && conda clean -afy \
    && find /opt/conda/ -follow -type f -name '*.pyc' -delete
CMD [ "/bin/sh" ]
```

Use a separate `environment.yml` file to define your Python environment. This avoids modifying the `Dockerfile` manually each time you need a new Python package. Create `containers/environment.yml`:

```
name: base
channels:
  - conda-forge
  - base
dependencies:
  - jupyterlab=1.0
```

To launch JupyterLab, first add a new step to your workflow in `wf.yml`

```
- id: "notebook"
  uses: "./containers/"
  args: ["jupyter", "--version"]
  options:
    ports:
      8888/tcp: 8888
```

Notes:

- `uses` is set to `./containers/` which tells Popper where to find the `Dockerfile` defining the container used for this step
- `ports` is set to `{8888/tcp: 8888}` which is necessary for the host machine to connect to the Jupyter Lab server in the container

Next, in the local command line, execute the `notebook` step in interactive mode:

```
popper sh -f wf.yml notebook
```

Now, in the Docker container's command line:

```
jupyter lab --ip 0.0.0.0 --no-browser --allow-root
```

Skip this second step if you only need the shell interface.

Notes:

- `--ip 0.0.0.0` allows the user to access JupyterLab from outside the container (by default, Jupyter only allows access from `localhost`).
- `--no-browser` tells jupyter to not expect to find a browser in the docker container.
- `--allow-root` runs JupyterLab as a root user (the recommended method for running Docker containers), which is not enabled by default.

Open the generated link in a browser to access JupyterLab.

Using other container engines

The above steps are for Docker. If you use Singularity, omit

```
options:
  ports:
    8888/tcp: 8888
```

Which is not needed because Singularity has no network isolation

5.7.5 Package management

It can be difficult to guess in advance which software libraries are needed in the final workflow. Instead, update the workflow requirements as you go using one of the package managers available for Python.

conda

Conda is recommended for package management because it has better dependency management and support for compiled libraries. When executing the `notebook` step interactively, install package as needed using (the easiest way to access the container's command line in this situation is Jupyter Lab's terminal interface):

```
conda install PACKAGE [PACKAGE ...]
```

Update the environment requirements with:

```
conda env export > containers/environment.yml
```

On the next use of the Docker image, Popper will rebuild it with the updated requirements (Note: this is triggered by`COPY environment.yml` in the `Dockerfile`).

pip

You can adapt the process decribed for `conda` to `pip`:

```
pip install PACKAGE [PACKAGE ...]
pip freeze > containers/requirements.txt
```

Modify the run command `RUN` in the `Dockerfile` to:

```
RUN pip install -r requirements.txt
```

Seperating docker images

Some workflows have conflicting software requirements between steps, for instance if two steps require different versions of a library. In this case, organize your container definitions as follows:

```
└── containers
    ├── step_A
    |   ├── Dockerfile
    |   └── environment.yml
    └── step_B
        ├── Dockerfile
        └── environment.yml
```

Then, in `wf.yml`:

```
- id: "step_A"
  uses: "./containers/step_A/"
# ...

- id: "step_b"
  uses: "./containers/step_B/
```

5.7.6 Models and visualization

Following the above, automate the other steps in your workflow using Popper. This section shows examples for:

- fitting a model to data
- generating model evaluation plots
- using the model to make predictions on a hold-out dataset

A first file, `src/models.py` defines the model this workflow uses:

```
from sklearn import impute, preprocessing, compose, pipeline, linear_model,␣
↪multioutput

def _get_preprocessor(num_features , cat_features):

    num_transformer = pipeline.Pipeline([
        ("scale", preprocessing.StandardScaler()),
        ("impute", impute.KNNImputer(n_neighbors = 10)),
    ])

    cat_transformer = pipeline.Pipeline([
        ("impute", impute.SimpleImputer(strategy = "constant", fill_value = "missing
↪")),
        ("encode", preprocessing.OneHotEncoder(drop = "first")),
```

(continues on next page)

(continued from previous page)

```
    ])

    preprocessor = compose.ColumnTransformer(
        [("num", num_transformer, num_features),
        ("cat", cat_transformer, cat_features)
    ])
    return preprocessor

def get_lr_model(num_features, cat_features, C = 1.0):

    model = pipeline.Pipeline([
        ("pre", _get_preprocessor(num_features, cat_features)),
        ("model", multioutput.MultiOutputClassifier(
                    linear_model.LogisticRegression(penalty="l1", C = C, solver =
↪"saga")
        )),
    ])
    return model
```

A second script, `src/predict.py`, uses this model to generate the predictions on the hold-out dataset:

```
import pandas as pd
import os
from models import get_lr_model

DATA_PATH = "data/raw"
PRED_PATH = "results/predictions"

if __name__ == "__main__":

   X_train = pd.read_csv(os.path.join(DATA_PATH, "training_set_features.csv")).drop(
       "respondent_id", axis = 1
   )

   X_test = pd.read_csv(os.path.join(DATA_PATH, "test_set_features.csv")).drop(
       "respondent_id", axis = 1
   )

   y_train = pd.read_csv(os.path.join(DATA_PATH, "training_set_labels.csv")).drop(
       "respondent_id", axis = 1
   )

   sub = pd.read_csv(os.path.join(DATA_PATH, "submission_format.csv"))

   num_features = X_train.columns[X_train.dtypes != "object"].values
   cat_features = X_train.columns[X_train.dtypes == "object"].values

   model = get_lr_model(num_features, cat_features, 1)
   model.fit(X_train, y_train)
   preds = model.predict_proba(X_test)

   sub["h1n1_vaccine"] = preds[0][:, 1]
   sub["seasonal_vaccine"] = preds[1][:, 1]
   sub.to_csv(os.path.join(PRED_PATH, "baseline_pred.csv"), index = False)
```

Add this script as a step in the Popper workflow. This must come after the `get_data` step

```
- id: "predict"
  uses: "./containers/"
  args: ["python", "src/predict.py"]
```

Notes:

- This use the same container as in the `notebook` step. Again, the final, 'canonical' analysis should be developed in the same environment as exploratory code.

Similarly, add `src/evaluate_model.py`, which generates model performance plots, to the workflow.

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np
import os
import pandas as pd
import seaborn as sns
from sklearn.model_selection import cross_val_score
from models import get_lr_model

DATA_PATH = "data/raw"
FIG_PATH = "output/figures"

if __name__ == "__main__":
    mpl.rcParams.update({"figure.autolayout": True, "figure.dpi": 150})
    sns.set()

    X_train = pd.read_csv(os.path.join(DATA_PATH, "training_set_features.csv")).drop(
        "respondent_id", axis=1
    )
    y_train = pd.read_csv(os.path.join(DATA_PATH, "training_set_labels.csv")).drop(
        "respondent_id", axis=1
    )

    num_features = X_train.columns[X_train.dtypes != "object"].values
    cat_features = X_train.columns[X_train.dtypes == "object"].values

    Cs = np.logspace(-2, 1, num = 10, base = 10)
    auc_scores = cross_val_score(
        estimator = get_model(num_features, cat_features, C),
        X = X_train,
        y = y_train,
        cv = 5,
        n_jobs = -1,
        scoring = "roc_auc",
    )

    fig, ax = plt.subplots()
    ax.plot(Cs, auc_scores)
    ax.vlines(
      Cs[np.argmax[auc_scores]],
      ymin = 0.82,
      ymax = 0.86,
      colors = "r",
      linestyle = "dotted"
    )
    ax.annotate(
      "$C = 0.464$ \n ROC AUC ={:.4f}".format(np.max(auc_scores)),
```

(continues on next page)

(continued from previous page)

```
      xy = (0.5, 0.835)
    )
    ax.set_xscale("log")
    ax.set_xlabel("$C$")
    ax.grid(axis = "x")
    ax.legend(["AUC", "best $C$"])
    ax.set_title("AUC for different values of $C$")
    fig.savefig(os.path.join(FIG_PATH, "lr_reg_performance.png"))
```

Use a similar step to the previous one:

```
- id: "figures"
  uses: "./"
  args: ["python, src/evaluate_model.py"]
```

Notes:

These steps each read data from `data/` and output to `results/`. It is good practice to keep the input and outputs of a workflow separate to avoid accidently modifying the original data, which is considered immutable.

5.7.7 Building a LaTeX paper

Wrap the build of the paper in your Popper workflow. This is useful to ensure that the pdf is always built with the most up-to-date data and figures.

```
- id: "paper"
  uses: "docker://blang/latex:ctanbasic"
  args: ["latexmk", "-pdf", "paper.tex"]
  dir: "/workspace/paper"
```

Notes:

- This step uses a basic LaTeX installation. For more sophisticated needs, use a full TexLive image
- `dir` is set to `workspace/paper` so that Popper looks for and outputs files in the `paper/` folder

5.7.8 Conclusion

This is the final workflow:

```
steps:
- id: "dataset"
  uses: "docker://jacobcarlborg/docker-alpine-wget"
  args: ["sh", "src/get_data.sh", "data"]

- id: "notebook"
  uses: "./"
  args: ["jupyter", "--version"]
  options:
    ports:
      8888/tcp: 8888

- id: "predict"
  uses: "./"
```

(continues on next page)

(continued from previous page)

```
  args: ["python, src/predict.py"]

- id: "figures"
  uses: "./"
  args: ["python, src/evaluate_model.py"]

- id: "paper"
  uses: "docker://blang/latex:ctanbasic"
  args: ["latexmk", "-pdf", "paper.tex"]
  dir: "/workspace/paper"
```

And this is the final project structure:

```
├──LICENSE
├── README.md               <- The top-level README.
├── wf.yml                  <- Definition of the workflow.
├── containers
|   ├── Dockerfile          <- Definition of the OS environment.
|   └── environment.yml     <- Definition of the Python environment.
├── data                    <- The original, immutable data dump.
├── results
|   ├── models              <- Serialized models, predictions, model summaries.
|   └── figures             <- Graphics created during analysis.
├── paper                   <- Generated analysis as PDF, LaTeX.
|   ├── paper.tex
|   └── referenced.bib
└── src                     <- Python source code for this project.
    ├── notebooks           <- Jupyter notebooks.
    ├── get_data.sh         <- Script for downloading the original data dump.
    ├── models.py           <- Script defining models.
    ├── predict.py          <- Script for generating model predictions.
    └── evaluate_model.py   <- Script for generating model evaluation plots.
```

To re-run the entire workflow, use:

```
popper run -f wf.yml
```

5.8 Computational research with R and RStudio Server

This guide explains how to use Popper to develop and run reproducible workflows for computational research in fields such as bioinformatics, machine learning, physics or statistics. Computational research with R relies on complex software dependencies that are difficult to port across environments. In addition, a typical workflow involves multiple dependent steps which will be hard to replicate if not properly documented. Popper offers a solution to these challenges:

- Poppers abstracts over software environments with Linux containers.
- Poppers forces you to define your workflow explicetely such that it can be re-run in in a single command.

Popper thus provides an open-source alternative to managed solutions such as Code Ocean for reproducible computational research.

5.8.1 Pre-requisites

You should have basic knowledge of git, the command line and R (code snippets in this guide use the tidyverse libraries).

In addition, you should be familiar with the concepts introduced in the Getting Started section. This guide uses examples from machine learning but no prior knowledge of the field is required.

By default, this guide assumes that you use the Docker container engine, but highlights where the workflow will differ if you use another engine.

5.8.2 Getting started

The examples presented in this guide come from a workflow developed for the Flu Shot Learning research competition on Driven Data. This workflow shows examples of using Popper to automate common tasks in computational research with R:

- downloading data
- using R Markdown
- fitting/simulating a model using `tidymodels`
- visualizing the results with `ggplot2`
- building a LaTeX paper with up-to-date results

To help follow allong, see this repository with the final version of the workflow. To adapt the advice in this guide to your own project, get started with this Cookiecutter template for Popper.

Initial project structure

```
├── LICENSE
├── README.md               <- The top-level README.
├── data                    <- The original, immutable data dump.
├── output
|   ├── models              <- Serialized models, predictions, model summaries.
|   └── figures             <- Graphics created during analysis.
├── paper                   <- Generated analysis as PDF, LaTeX.
|   ├── paper.tex
|   └── referenced.bib
└── src                     <- R source code for this project.
    ├── notebooks           <- RMarkdown notebooks.
    ├── get_data.sh         <- Script for downloading the original data dump.
    ├── models.py           <- Script defining models.
    ├── predict.py          <- Script for generating model predictions.
    └── evaluate_model.py   <- Script for generating model evaluation plots.
```

5.8.3 Getting data

Your workflow should automate downloading or generating data to ensure that it uses the correct, up-to-date version of the data. In this example, you can download data with a simple shell script:

```
#!/bin/sh
cd $1

wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/test_set_features.csv" -
↪-no-check-certificate
```

(continues on next page)

(continued from previous page)

```
wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/training_set_labels.csv
↪" --no-check-certificate
wget "https://s3.amazonaws.com/drivendata-prod/data/66/public/training_set_features.
↪csv" --no-check-certificate

echo "Files downloaded: $(ls)"
```

Now, wrap this step using a Popper workflow. In a new file `wf.yml` at the root of the folder,

```
steps:
- id: "dataset"
  uses: "docker://jacobcarlborg/docker-alpine-wget"
  args: ["src/get_data.sh", "data"]
```

Notes:

- pick a Docker image that contains the necessary utilities. For instance, a default Alpine image does not include `wget`.

5.8.4 Using RStudio Server

This sections explains how to use Popper to launch RStudio Server, which provides a convenient environment for exploratory work. Refactoring successful experiments into your final workflow is easier if you keep the software environment consistent between both. Thus, you should do both your exploratory and "canonical" work in the same container.

To run RStudio Server, first add a new step to your workflow in `wf.yml`

```
- id: "rstudio"
  uses: "getpopper/r/verse:3.6.2"
  runs: ["r", "--version"]
  options:
    ports:
      8787: 8787
```

This step uses the `getpopper/r/verse` image. `getpopper` on Dockerhub hosts a library of Docker images configured to work well with Popper and RStudio.

Notes:

- `ports` is set to `{8787: 8787}` which is necessary for the host machine to connect
- the container is based by default on the Rocker `verse` image, which includes the `tidyverse` libraries and `latex`. If you do not plan on using `tidyverse` or Latex, using the `getpopper/R/rstudio` image (based on `rocker/rstudio`) will make for smaller images sizes

Go to `localhost:8787` in your browser to access RStudio Server. Log in with username and password `rstudio`.

Using other container engines

The above steps are for Docker. If you use Singularity, omit

```
options:
  ports:
    8787/tcp: 8787
```

Which is not needed because Singularity has no network isolation.

5.8.5 Package and image management

To manage project dependencies, you should use a fully container-based apporach. R provides a default dependency management throughs its packaging features, but are not well suited to pinning exact dependencies. While more modern alternatives exist (`packrat` and `renv`), both make assumptions that fit poorly into Popper workflows if you also want to use RStudio.

Instead, you should use containerit, a R package which automatically builds a Dockerfile from the packages loaded in the current environment.

For instance, this workflow uses the `tidyverse` and `tidymodels` libraries. The base Docker image used in the following does not include `tidymodels`, so it needs to be installed. In the RStudio Server prompt,

```
install.packages("tidymodels")
```

Furthermore, this workflow uses an optional `tidymodels` dependencies, `glmnet`, for fitting a regularized logistic regress

```
install.packages("glmnet")
```

Load containerit:

```
library(containerit)
```

Create a Dockerfile from the current R session

```
library(tidymodels)
library(tidyverse)
library(glmnet)
my_dockerfile <- containerit::dockerfile(
  image = "getpopper/r/verse:3.6.2",
  maintainer = "apoirel@ucsc.edu",
  container_workdir = NULL
)
```

Alternatively, if `src/` were already populated with the souce code for the project, it would be possible to create a Dockerfile for a set of files:

```
my_dockerfile <- containerit::dockerfile(from = "./src",
  image = "getpopper/r/verse:3.6.2",
  maintainer = "apoirel@ucsc.edu",
  container_workdir = NULL
)
```

Write the Dockerfile:

```
containerit::write(my_dockerfile)
```

This is the generated Dockerfile:

```
FROM getpopper/verse:3.6.2
LABEL maintainer="apoirel@ucsc.edu"
RUN ["install2.r", "dplyr", "forcats", "ggplot2", "purrr", "readr", "stringr", "tibble
↪", "tidyr", "tidyverse", "rsample", "parsnip", "recipes", "workflows", "tune",
↪"yardstick", "broom", "dials", "tidymodels", "glmnet"]
EXPOSE 8787
CMD ["R"]
```

At this point, you should change your workflow to use this Dockerfile with other steps using R. (`uses: ./`)

5.8.6 Models and visualization

Following the above, automate the other steps in your workflow using Popper. This section shows examples for:

- fitting a model to data
- generating model evaluation plots
- using the model to make predictions on a hold-out dataset

In this example, modeling is done using the `tidymodels` libraries.

A first file, `src/models.py` defines the data pre-processing steps the model will use:

```
library(tidyverse)
library(tidymodels)

get_preprocessor <- function(df_train, target, ignored) {
    df_train <- df_train %>% select(!ignored)
    rec <-
        recipe(as.formula(paste(target, "~ .")), data = df_train) %>%
        step_medianimpute(all_numeric()) %>%
        step_normalize(all_numeric(), -all_outcomes()) %>%
        step_unknown(all_nominal()) %>%
        step_dummy(all_nominal()) %>%
        step_num2factor(
          target,
          transform = function(x) as.integer(x + 1),
          levels = c("0", "1"),
          skip=TRUE
        )
    return(rec)
}
```

A second script, `src/predict.R`, uses this to generate the predictions on the hold-out dataset

```
library(tidyverse)
library(tidymodels)

DATA_PATH = "data"
OUTPUT_PATH = "output"

source("src/models.R")

df_train <- read_csv(paste(DATA_PATH, "training_set_features.csv", sep = "/"))
y_train <- read_csv(paste(DATA_PATH, "training_set_labels.csv", sep = "/"))
df_test <- read_csv(paste(DATA_PATH, "test_set_features.csv", sep = "/"))
df_submission <- read_csv(paste(DATA_PATH, "submission_format.csv", sep = "/"))

df_train <-
    left_join(df_train, y_train, on = "respondent_id", keep = FALSE) %>%
    select(!"respondent_id")

get_predictions <- function(target, ignored, df_train, df_test) {
    lr_model <-
        logistic_reg(penalty = 0.01, mixture = 1) %>%
        set_engine("glmnet")

    predictions <-
        workflow() %>%
```

(continues on next page)

(continued from previous page)

```
        add_recipe(get_preprocessor(df_train, target, ignored)) %>%
        add_model(lr_model) %>%
        fit(data = df_train) %>%
        predict(df_test, type = "prob") %>% # targets are probabilities
        pull(".pred_1") # we want the probability *being* vaccinated

    return(predictions)
}

preds_seasonal <-
    get_predictions("seasonal_vaccine", "h1n1_vaccine", df_train, df_test)

preds_h1n1 <-
    get_predictions("h1n1_vaccine", "seasonal_vaccine", df_train, df_test)

# save predictions to submission file
df_submission %>%
    mutate(h1n1_vaccine = preds_h1n1) %>%
    mutate(seasonal_vaccine = preds_seasonal) %>%
    write_csv(paste(OUTPUT_PATH, "submission.csv", sep = "/"))
```

As this as a set in the Popper workflow. This must come after the `get_data` step

```
- id: "predict"
  uses: "./"
  args: ["Rscript", "predict.R"]
```

Notes:

- This use the same container as in the `rstudio` step. Again, the final, 'canonical' analysis should be developed in the same environment as exploratory code.

Similary, add `src/evaluate_model.R`, which generates model performance plots, to the workflow

```
library(tidyverse)
library(tidymodels)

DATA_PATH = "data"
OUTPUT_PATH = "output"

source("src/models.R")

df_train <- read_csv(paste(DATA_PATH, "training_set_features.csv", sep = "/"))
y_train <- read_csv(paste(DATA_PATH, "training_set_labels.csv", sep = "/"))

df_train <-
    left_join(df_train, y_train, on = "respondent_id", keep = FALSE) %>%
    select(!"respondent_id")

get_cv_results <- function(df_train, target, ignored) {

    # define model
    lr_model <-
        logistic_reg(penalty = tune(), mixture = 1) %>%
        set_engine("glmnet")

    wf <-
```

(continues on next page)

(continued from previous page)

```
        workflow() %>%
        add_recipe(get_preprocessor(df_train, target, ignored)) %>%
        add_model(lr_model)

    # cv parameters
    folds <- df_train %>% vfold_cv(v = 5)
    lr_grid <-
        grid_regular(
            penalty(range = c(-2,1), trans = log10_trans()),
            levels = 10
        )

    # collect cv results
    cv_res <-
        wf %>%
        tune_grid(
            resamples = folds,
            grid = lr_grid,
            metric = metric_set(roc_auc)
        ) %>%
        collect_metrics()

    # plot_results
    cv_res %>%
    ggplot(aes(penalty, mean)) +
    geom_line(size = 1.2, color = "red", alpha = 0.5) +
    geom_point(color = "red") +
    scale_x_log10(labels = scales::label_number()) +
    scale_color_manual(values = c("#CC6666")) +
    ggtitle(expression(paste("AUC for different ", L[1], " penalties")))

    ggsave(
        paste("cv_", target, ".png", sep = ""),
        path = paste(OUTPUT_PATH, "figures", sep = "/")
    )
}

get_cv_results(df_train, "h1n1_vaccine", "seasonal_vaccine")
get_cv_results(df_train, "seasonal_vaccine", "h1n1_vaccine")
```

```
- id: "figures"
  uses: "./"
  args: ["Rscript", "evaluate_model.R"]
```

Note that these steps each read data from `data/` and output to `output/`. It is good practice to keep the input and outputs of a workflow separate to avoid accidently modifying the original data, which is considered immutable.

5.8.7 Building a PDF paper

Wrap the build of the final paper or report in your Popper workflow. This is useful to ensure that the pdf is always built with the most up-to-date data and figures.

Latex

This is the step for building a LaTeX paper. Note we use the same image as in previous steps since `rocker/verse` includes a full LaTeX installation.

```
- id: "paper"
  uses: "./"
  args: ["latexmk", "-pdf", "paper.tex"]
  dir: "/workspace/paper"
```

RMarkdown

Many R users find it more convenient to write up the final analysis directly in RMarkdown and then knit the document to HTML or pdf. You can easily modify the above step to support this workflow.

```
- id: "paper"
  uses: "./"
  args: ["R", "-e", "library(rmarkdown);rmarkdown::render("paper/paper.Rmd", output_
↪format="all")"]
  dir: "/workspace/paper"
```

5.8.8 Conclusion

This is the final workflow, assuming the paper is written in LaTeX

```
steps:
- id: "dataset"
  uses: "docker://jacobcarlborg/docker-alpine-wget"
  args: ["sh", "src/get_data.sh", "data"]

- id: "rstudio"
  uses: "./"
  args: ["rstudio-server", "start"]
  options:
    ports:
      8787: 8787

- id: "figures"
  uses: "./"
  args: ["Rscript", "evaluate_model.R"]

- id: "predict"
  uses: "./"
  args: ["Rscript", "predict.R"]

- id: "paper"
  uses: "./"
  args: ["latexmk", "-pdf", "paper.tex"]
  dir: "/workspace/paper"
```

And this is is the final project structure

```
├── LICENSE
├── README.md                <- The top-level README.
```

(continues on next page)

(continued from previous page)

```
├── wf.yml                    <- Definition of the workflow.
├── Dockerfile                <- Dockerfile used by the workflow.
├── data                      <- The original, immutable data dump.
├── output
|   ├── models                <- Serialized models, predictions, model summaries.
|   └── figures               <- Graphics created during analysis.
├── paper                     <- Generated analysis as PDF, LaTeX.
|   ├── paper.tex             <- LaTeX source for the paper.
|   └── referenced.bib
└── R                         <- R source code for this project.
    ├── notebooks             <- Exploratory Rmarkdown notebooks.
    ├── get_data.sh           <- Script for downloading the original data dump.
    ├── models.R              <- Script defining models.
    ├── predict.R             <- Script for generating model predictions.
    └── evaluate_model.R      <- Script for generating model evaluation plots.
```

CHAPTER 6

Other Resources

- A list of example workflows can be found at https://github.com/popperized/popper-examples.
- Self-paced hands-on tutorial.

CHAPTER 7

FAQ

7.1 How can I create a virtual environment to install Popper

The following creates a virtual environment in a `$HOME/venvs/popper` folder:

```
# create virtualenv
virtualenv $HOME/venvs/popper

# activate it
source $HOME/venvs/popper/bin/activate

# install Popper in it
pip install popper
```

The first step is is only done once. After closing your shell, or opening another tab of your terminal emulator, you'll have to reload the environment (`activate it` line above). For more on virtual environments, see here.

7.2 How can we deal with large datasets? For example I have to work on large data of hundreds GB, how would this be integrated into Popper?

For datasets that are large enough that they cannot be managed by Git, solutions such as a PFS, GitLFS, Datapackages, ckan, among others exist. These tools and services allow users to manage large datasets and version-control them. From the point of view of Popper, this is just another tool that will get invoked as part of the execution of a pipeline. As part of our documentation, we have examples on how to use datapackages, and another on how to use data.world.

7.3 How can Popper capture more complex workflows? For example, automatically restarting failed tasks?

A Popper pipeline is a simple sequence of "containerized bash scripts". Popper is not a replacement for scientific workflow engines, instead, its goal is to capture the highest-most workflow: the human interaction with a terminal.

7.4 Can I follow Popper in computational science research, as opposed to computer science?

Yes, the goal for Popper is to make it a domain-agnostic experimentation protocol. See the https://github.com/popperized/popper-examples repository for examples.

7.5 How to apply the Popper protocol for applications that take large quantities of computer time?

The `popper run` takes an optional `STEP` argument that can be used to execute a workflow up to a certain step. Run `popper run --help` for more.

CHAPTER 8

Contributing

Read the CONTRIBUTING.md file contained in the main repository.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search