Jikken Documentation

Release 0.2.1

Agis Oikonomou

Dec 05, 2017

Contents:

Jikken is a lightweight cli experiment manager for scientific experiments written in python.

It makes very few assumptions on the experiment and requires almost zero code changes to the scripts. In fact the only assumption is that the main function of the script to be run accepts a positional argument with either a directory with json/yaml config files or a path to single json/yaml file. That's it. Optionally if the code of the experiment is in a git repo, the git info will be added as well.

CHAPTER 1

Features

- Python 3.{5,6} code
- Support for TinyDB, MongoDB, and ES
- tagging of experiments
- CLI to access experiment data
- only requires the script to load the variables from a file or folder
- support for json/yaml configs
- understands git directories

CHAPTER 2

Installation

Jikken can be installed from pip. This installation is compatible with Linux/Mac OSX and Python 3.5 or greater is required. To install the package use:

or if the default python on your system is 2.7 use instead

```
pip3 install jikken
```

CHAPTER 3

Usage Example

The main goal of *Jikken* is to require as little changes to your experiment code as possible. It's main assumption is that you have a script that runs an experiment and that the scripts accepts at least a positional argument with the location of the experiment's config. e.g. Let's assume you have an experiment runs by running my_experiment.py and it reads its configuration from a yaml file myconfig.yaml you would run it with:

```
python my_experiment.py myconfig.yaml
```

Then in order to let jikken record the experiment you would run instead:

```
jikken run my_experiment.py -c myconfig.yaml -n "my first experiment"
```

Jikken will then capture the config, Start the experiment, capture stdout and stderr, and save the information in the database as set in the config file.

CHAPTER 4

Documentation

The full documentation of jikken can be found here

4.1 Contributing

1. Fork it here
2. Create your feature branch (*git checkout -b feature/fooBar*)
3. Commit your changes (*git commit -am 'Add some fooBar'*)
4. Push to the branch (*git push origin feature/fooBar*)
5. Create a new Pull Request

4.2 Release History

- **0.2.1**
  - Fixed bug with updated monitored (and added test)
  - Added cli delete command for both experiments and multistage experiments
  - refactored documentation a bit
- **0.2.0**
  - Added mongodb and ES support
  - Added support for multistage experiments
  - started writing documentation on read the docs
- **0.1.0**
  - Work in progress

4.3 Versioning

We use SemVer for versioning. For the versions available, see the tags on this repository

4.4 Authors

- **Agis Oikonomou** - *Initial work*

4.5 License

This project is licensed under the MIT License - see the LICENSE file for details

4.6 Acknowledgments

- Brian Okken and his great book on python testing and a great influence on the structure of the code: Python Testing with Pytest.
- Francois Chollet and his book Deep Learning with Python. The examples of jikken are all based on the ones from the book.

4.6.1 Installation

Install from Source

Jikken can be installed from source. This installation is compatible with Linux/Mac OS X and Python *3.{5,6}*. In order to use mongoDB or elasticsearch a database need to be setup and running already.

- git clone the library

```
git clone https://github.com/outcastofmusic/jikken.git
```

- run the setup install

```
python3 setup.py install
```

Database Configuration

Jikken uses a configuration file named *config* with the database information it will use. It will look for it in the experiment scripts directory in teh file *$SCRIPT_DIR/.jikken/config*. If it can't find a project specific config it will look for a global one inside the users HOME dir in the file *~/.jikken/config*. If That doesn't exist either it will create it with default values using tinydb as the database.

The *config* file should look like this

```
[db]
path = ~/.jikken/jikken_db/
type = tiny
name = jikken
```

Where: - path is the path to the database (or a valid *uri* in the case of mongodb or es. - type is the type of the database (valid options are: *tiny*, *mongo*, *es* (ES not implemented yet). - name is the name of the database to use. Default is *jikken*.

An example of a config file using *Mongo* would be

```
[db]
path = mongodb://localhost:27017
type = mongo
name = jikken
```

An example of a config file using *ElasticSearch* would be

```
[db]
path = http://localhost:9200
type = es
name = jikken
```

Setup MongoDB using docker

pull the mongodb image:

```
docker pull mongo
```

then run the image. You can use the *-p* flag to map the port to a localport (that matches the one in the config file and also mount a local folder where the db will be located

```
docker run --name jikken-mongo -p $LOCALPORT:27017 -v $LOCALPATH:/data/db -d mongo
```

For more info see the official docker mongo information.

Setup ES using docker

pull the es image:

```
docker pull docker.elastic.co/elasticsearch/elasticsearch-oss:6.0.0
```

then run the image. (this command is for development mode):

```
docker run -p 9200:9200 -p 9300:9300 -e "discovery.type=single-node" docker.elastic.
↪co/elasticsearch/elasticsearch-oss:6.0.0
```

For more info see the official docker es guide.

4.6.2 Usage

Running an Experiment

The subcommand to run experiments is *run*. If we type:

```
jikken run -h
```

we get the following information about the command:

```
Usage: jikken run [OPTIONS] SCRIPT_PATH

  run a single stage experiment from a script. e.g. jikken run script.py -c
    config.yaml

    Options:
      -c, --configuration_path PATH  A file or a directory with files that hold
                                     the variables that define the experiment
                                     [required]
      -n, --name TEXT                the experiment name  [required]
      -r, --ref_path PATH            A file or a directory with files that hold
                                     the variables that define the experiment
      -a, --args TEXT                extra arguments that can be passed to the
                                     script multiple can be added,e.g. -a a=2 -a
                                     batch_size=63 -a early_stopping=False
      -t, --tags TEXT                tags that can be used to distinguish the
                                     experiment inside the database. Multiple can
                                     be added e.g. -t org_name -t small_data -t
                                     model_1
      -h, --help                     Show this message and exit.
```

jikken run expects a positional argument which is the path of the script to run and a number of options. All options have both a short form and a long form.

Besides the script path two options are required. The first one is *-c*, *–configuration_path* and should be followed by the path to the configuration for the experiment. This can be either the path to a *.json* or *.yaml* file or the path to a directory holding multiple files. The files in the directory can be organized as subfolders. Of course as explained above this path will be passed as positional input to the script itself, and the script should be able to use it. The second required option is *-n*, *–name*. Itt should be a small name way to identify the experiment to be run. It doesn't have to be unique. Multiple words can be used e.g. `-n "my first nlp experiment"` and can later be retrieved from the db by searching for part of the name.

Optionally the *-t*, *–tag* option can be used to add tags to the experiment. Multiple tags can be added to an experiment e.g.

```
jikken run my_experiment.py -c myconfig.yaml -n "my first experiment" -t nlp -t␣
↪tensorflow -t kaggle_data
```

Tags are pivotal for distinguishing between experiments and their generous use is recommended.

In case the script expects more keyword arguments the *-a*, *–args* option can be used to pass them. These will also be stored in the database as parts of the variables used. e.g

```
jikken run my_experiment.py -c myconfig.yaml -n "my first experiment" -a batch_
↪size=15 -a early_stopping=False
```

Extra positional arguments are currently not supported.

Using a reference config

Sometimes you have some reference configuration and you just want to change one or two options. The *-r*, *–ref_path* option allows you to do that. When *-r* option is used, whatever's afterward is used as the reference and what is afterh the *-c* option will be used to update the reference variables. The structure for the update variables must match the structure of the reference variables, but only the variables that will be updated need to inside the *-r* path. e.g.

```
jikken run my_experiment.py -c update_config.yaml -r myconfig.yaml -n "my first␣
↪experiment"
```

where myconfig.yaml could be

```
model_parameters:
   num_layers: 10
   hidden_size: 50
   optimizer: Adam
input_parameters:
   batch_size: 128
   augment: true
```

and then update_config.yaml need only be

```
model_parameters:
   optimizer: SGD
input_parameters:
   batch_size: 64
```

Similarly if *-r* is a directory, then *-c* must also be a directory with the files with updated variables need to match the relative paths of those in the reference directory.

Monitoring Variables

Jikken allows you to monitor variables as the code is executed and store their value as your experiments runs. In order to do this you need to import the log_value function from jikken. and then pass it it the value to be monitored e.g.

```
from jikken import log_value
for epoch in range(100):
    loss = fancy_experiment()
    if epoch % 10 == 0:
            log_value("loss",experiment)
```

The above example will log the value of the loss every 10 epochs. *log_value()* can also be used with callback fuctionor hooks (see examples) that call it when it is required to log a value.

Running Multistage Experiments

Sometimes an experiment is too complicated and can be split into different stages. For example a multistage experimentwith three steps coul be designed as follows. - The first stage would convert input data to features. - The second stages gets the features and trains a model - The third stage tests the trained model on a test set

By splitting an experiment in stages like this allows some stages to remain the same while changing other stages, e.g.train many different models with the same features. It also allows for segregating experiment info makin git much easier to check on data afterwards.

Jikken allows that with the stage subcommand,i.e. *jikken stage run '. Running 'jikken stage run -h* gives us:

```
Usage: jikken stage run [OPTIONS] SCRIPT_PATH

  run a stage of a multistage experiment from a script. e.g. jikken run
    script.py -c config.yaml

    Options:
```

```
  -i, --input_dir DIRECTORY
  -o, --output_dir DIRECTORY      [required]
  -c, --configuration_path PATH   A file or a directory with files that hold
                                  the variables that define the experiment
                                  [required]
  -n, --name TEXT                 the experiment name  [required]
  -s, --stage_name TEXT           the stage name  [required]
  -r, --ref_path PATH             A file or a directory with files that hold
                                  the variables that define the experiment
  -a, --args TEXT                 extra arguments that can be passed to the
                                  script multiple can be added,e.g. -a a=2 -a
                                  batch_size=63 -a early_stopping=False
  -t, --tags TEXT                 tags that can be used to distinguish the
                                  experiment inside the database. Multiple can
                                  be added e.g. -t org_name -t small_data -t
                                  model_1
  -h, --help                      Show this message and exit.
```

jikken stage run uses the same positional argument *SCRIPT_PATH* and has a lot of common options with *jikken run*. The main difference is the addition of three more options:

The first is the *-i*, *–input_dir* option. This holds the location of the input dir to the experiment and is not required as the first stage might not have require an input dir. The *-o*, *–output_dir* option respectively, is where the output of the stage will be stored. This directory should be used as the *-i* option of the subsequent step. Jikken will use those directories to store metadata in order to keep track of how the different stages relate to each other. An *-o* is required at every stage for this reason. Finally the *-s*, *–stage_name* option should be text that describes the specific stages. An example of this stage could be the following:

```
jikken stage my_experiment_preprocessing.py -c myconfig_preprocessing.yaml -n "my␣
↪first experiment" -s "preprocessing" -o processing_results_dir
jikken stage my_experiment_training.py -c myconfig_training.yaml -n "my first␣
↪experiment" -s "training" -t "svm" -i processing_results_dir -o trained_model_dir
jikken stage my_experiment_validation.py -c myconfig_validation.yaml -n "my first␣
↪experiment" -s "validation" -i trained_model_dir -o validation_results_dir
```

Resuming An Experiment

As stages of multistage experiments have outputs this can be used to load the files from an output and resume a single stage. E.g. A training stage that has trained and saved the models weights to an output folder can load the model from the output folder and continue training. this can be done by using the *jikken stage resume* command

Retrieving Experiments from the database

Jikken allows you to retrieve information about experiments using the *jikken list* subcommands:

- *jikken list tags* retrieves a list of all tags in the database
- *jikken list count* returns the number of all experiments in the database, (optinally maching names and/or tags)
- *jikken list exp* allows you to query the database for any experiment run, including stages of multistage experiments.
- *jikken list mse* allows you to query the database for multistage experiments.
- *jikken list best* returns the experiment that has the best, (min or max) value of some metric from the query

jikken list tags

jikken list tags is the simpler of all of the above subcommands. It has no arguments or options and simply returns a list of all tags in the database.

jikken list count

jikken list count can be used to return the number of experiments in the database. This includes all experiments including mse stages. Running *jikken list count* without arguments returns everything in the db. *jikken list count* has three optional arguments:

- *-t*, *–tags* can be used to provide a number of tags jikken will try to match
- *-q*, *–query* is used in conjunction with the tags options. It can take two values *all* or *any*. *all* will try to match all provided tags, while *or* will try to provide any of the provided tags and any of the provided names.
- *-n*, *–name* can be used to provide a number of names jikken will try to match. Jikken will also match parts of names

jikken list exp

jikken list exp is the main command for retrieving experiment information. It has a plethora of options allowing for many different queries as well as how the retrieved data is presented.

Running *jikken list exp -h* returns the following

```
Usage: jikken list exp [OPTIONS]

  (Experiments): list experiments

Options:
  -i, --ids TEXT                  the ids to print
  -t, --tags TEXT                 the tags that need to be matched
  -n, --names TEXT                experiment names that need to be matched
  -s, --schema TEXT               hash that matches experiment schema hash
  --status [running|error|interrupted|completed]
                                  status of the experiment
  -p, --param_schema TEXT         hash that matches the experiment schema with
                                  parameters hash
  -q, --query [all|any]           the type of query ot be used (all|any)
  --stdout / --no-stdout          print the stdout of the experiments listed
  --stderr / --no-stderr          print the stderr of the experiments listed
  --var / --no-var                print the configuration variables of the
                                  experiments listed
  --git / --no-git                print git information of the experiments
                                  listed
  --monitored / --no-monitored    print monitored variables of the experiments
                                  listed
  -h, --help                      Show this message and exit.
```

The options for list can be split into two categories. Options that affect the query to the database and options that affect how the results are presented in stdout.

Query Options

Each experiment added to the database is assigned a unique id. This id is different depening on the type of underlying database.

- *-i*, *-ids*, allows for explicitly retrieving specific ids. When the *-i* option is used no other option is taken into account to formulate the query. Multiple ids can be retrieved by using the option multiple times e.g.

```
jikken list env -i id1 -i id2 -i id3
```

- *-t*, *–tags*. The tags option is used to retrieve experiments with matching tags. Multiple tags can be added by using the option multiple times. e.g.

```
jikken list env -t tag1 -t tag2 -t tag3
```

- *-q*, *–query*. The query option is used in conjunction with the tags option. It can be set to *all* (Default) or *any*. The former query matching all tags provided while the latter any tags provided. e.g.

```
jikken list env -t tag1 -t tag2 -t tag3 -q any
```

- *-n*, *–names*. The names option is used to retrieve experiments with matching names. Multiple names can be added to the query by using the option multiple times. e.g.

```
jikken list env -n "my first experiment" -n "second"
```

- *-s*, *–schema*. The schema option is about a hash jikken creates based on the schema of an experiment. The hash takes into account the variables used to configure the experiment, but **not** their actual values, and the commit of the script repo if it is a git repo‘. So for example if you run different experiments of the same model doing a hyperparameter search. All experiments will have the same schema hash, as the **schema** of the experiment remains the same. This option can be used multiple times as well. e.g.

```
jikken list env -s hash1 -s hash2
```

- *-p*, *–param_schema*. The param_schema option is about a hash jikken creates similar to the schema hash mentioned above, but this one takes into account the values of the variables as well. Hence experiments with the same param_schema will have been run with the exact same configuration and the exact same code. This option can be used multiple times as well. e.g.

```
jikken list env -p phash1 -p phash2
```

- *–status*. The status option queries on the status parameter of an experiment. The status parameter is created by jikken whenever a new experiment is added. It can be any of the following:
  - *created*: The experiment database entry has been created but the code has not been run yet.
  - *running*: The experiment is currently running
  - *completed*: The experiment has completed successfully
  - *interrupted*: The experiment did not complete successfully because it was interrupted by the user. e.g. using Ctrl-C to stop it.
  - *error*: The experiment did not complete successfully due to an error/exception while it was trying to run.

As with the previous options this can be queried with multiple values e.g the following will return all experiments with status error or interrupted.

```
jikken list env --status error --status interrupted
```

Print Options

jikken list best

jikken list mse

Coding a script that works with jikken

4.6.3 Indices and tables

- genindex
- modindex
- search