racket Documentation

Release 0.2.0

Carlo Mazzaferro

Nov 18, 2018

Contents:

CHAPTER 1

Installation

1.1 Requirements

`racket` is tested on python 3.6. It won't work in 3.7, as there is no TensorFlow release for this python version, and it probably won't work either on 3.5 since I use f-strings extensively.

Contributions are more than welcome to make the project compatible with other python versions!

`docker` and `docker-compose` are also required. Reasonably up-to-date versions should suffice.

1.2 Stable release

To install racket, run this command in your terminal:

```
$ pip install racket
```

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

If you don't have pip installed, this Python installation guide can guide you through the process.

1.3 From sources

The sources for racket can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/carlomazzaferro/racket
```

Or download the tarball:

```
$ curl  -OL https://github.com/carlomazzaferro/racket/tarball/master
```

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

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

CHAPTER 2

Concepts

The project is built with the premise of getting to a working prototype quickly without sacrificing flexibility.

This is achieved by leveraging the full capabilities of TFS while enabling the user to interact with a simple, expressive API.

2.1 Model Versioning

TFS has as one of its features model discovery. Namely, if a new model gets persisted in the directory where models have been specified to live (done usually in a Docker file (see here for more details), and the subdirectory of the model path has a number strictly greater than the existing folders, it will automatically load it.

Although useful, this formulation is clearly quite inflexible. If you want to modify the discovery scheme, you'd have to fiddle with TFS's C++ API. If instead you'd like to roll back models, you may have to create a new directory or modify the existing ones.

Racket instead abstracts away TFS the versioning schematics and allows the user to simply define the model version when instantiating a new model, while allowing the user to any specific version with a single cli command.

Model versions in racket follow semantic versioning and are supplied to it as strings.

2.2 Learners

Apart from the CLI and the RESTful access layer, most of the public API revolves around the Learner set of classes. These essentially define the patterns of interaction between the code that generates the models and the filesystem/database.

The idea is letting the user define the core of their needs (i.e., the learner's architecture) while getting for free the storage, versioning, and serving capabilities.

2.3 A Word on Persistence

The project relies heavily on SQLAlchemy to manage model metadata. I've found it to greatly reduce complexity when managing a relatively high number of models, as it makes it extremely easy to query for, and reason about the existing models. As a default, the project will use SQLite as its default data store, but that can be changed very easily by changing the configuration and specifying a database backend in the `racket.yaml` file, which gets generated automatically once the `racket init` command gets called.

CHAPTER 3

Usage

While the DemoVideo provides a quick overview of the functionality of `racket`, this document will explain in detail the steps of the demo, and provide resources to learn more about the inner workings of the project.

3.1 Starting a New Project

From the command-line:

```
racket init --name project-name --path path/to/directory
```

Will create a directory named `project-name` in the the specified path with all the required files to start serving models. Of particular note, the directory will have the following files:

```
docker-compose.yaml
Dockerfile
racket.yaml
regression.py
classification.py
.gitignore
```

To start TensorFlow Serving (TFS), run:

```
docker-compose up --build
```

Add the `-d` flag if you'd like to run it on the background.

3.2 Serving Your First Model

To create a new model, you can edit the `classification.py` or the `regression.py`. Those files define very basic Keras models, but they have a few quirks. Namely, the class definition inherits from the class*`racket.KerasLearner`*, which provides built in functionality to store models in a suitable format for TFS, as well as functionality to store metadata and historical scores of the model. The method *`racket.KerasLearner.store()`* is responsible for this functionality.

3.2.1 The `KerasLearner` Base Class

In order to user the class as your base class, when you create a class that inherits from it you must define a few things. Namely, it must have the following attributes and methods implemented:

Required attributes:

- `VERSION`: a string of the form `'major.minor.patch'`
- `MODEL_TYPE`: a string specifying what kind of model it is (e.g. a regression or classification) a string specifying the model's name
- `MODEL_NAME`: a string specifying the model's name

Required methods:

- `fit(x, y, x_val, y_val, *args, **kwargs)`: a method that specifies how to fit the model
- `build_model()`: a method that specifies how to compile the model

That's all. Having done that, you can call `fit()` as you normally would, after which you can call `store()`, which will take care of all the wiring needed to version, serve, serialize, and expose the model.

Refer to *`racket.KerasLearner`* for more information about the inner workings of the methods implemented, and how the inner workings are leveraged to interact with TFS

CHAPTER 4

RESTful Acesss

Run:

```
$ racket dashboard
```

And point your browser to http://0.0.0.0:8000/api/v1, where you'll find the documented API.

4.1 Infer

TODO: Document this

4.2 Discover

TODO: Document this

CHAPTER 5

Module Index

5.1 Racket's CLI

5.1.1 racket

racket CLI tool to:

- Create new projects
- Interact with racket server.
- Manage model lifecycle

Check the help available for each command listed below.

```
racket [OPTIONS] COMMAND [ARGS]...
```

Options

-v, --verbose
Turn on debug logging

dashboard

```
racket dashboard [OPTIONS]
```

Options

-h, --host `<host>`
Host on which to server

-p, --port <port>
Port on which to expose app

-e, --env <env>
Environment (dev, test, or prod)

-c, --clean
Clean up database

init

Creates a new project

```
racket init [OPTIONS]
```

Options

--name <name>
Name of the project

--path <path>
Directory where the new project will be created

ls

List available models, filtering and sorting as desired

Running:

```
$ racket ls -a  # returns the active model's metadata
```

Will return:

```
  model_id  model_name      major    minor    patch    version_dir  active    created_
↪at                   model_type    scoring_fn            score
----------  ------------  --------  --------  --------  -------------  --------  --------
↪-------------------  ------------  ------------------  -------
         1  base              0        1        0              1  True      2018-11-
↪14 22:53:52.455635  regression    loss               9378.25
         1  base              0        1        0              1  True      2018-11-
↪14 22:53:52.455635  regression    mean_squared_error  9378.25
```

```
racket ls [OPTIONS]
```

Options

-n, --name <name>
List available models with a specific name

-v, --version <version>
Retrieve modles of only a specific version, e.g. M1, m2, or p1 (M: Major, m: minor, p: patch

-t, --type <m_type>
Filter on model type

-a, --active
Returns currently active model

--id <model_id>
Filters on model id

serve

Serve a specific model.

This allows you to specify either a model-id or a the name + version of a specific model that you'd like to serve. If the model-id is specified, the name and versions are ignored.

Throws an error if the specified model do not exist.

```
racket serve [OPTIONS]
```

Options

--model-id <model_id>
Model unique identifier

--model-name <model_name>
Model name

--version <version>
Model version as major.minor.patch, or latest

v

Retrive the version of the current `racket` install

```
racket v [OPTIONS]
```

version

Retrive the version of the current `racket` install

```
racket version [OPTIONS]
```

5.2 Internals

class racket.**Learner**
Abstract Base Class for any learner implemented (currently Keras only, but more are planned).

Note: This as an abstract class and cannot be instantiated

semantic
str – Semantic representation of the model version

major
int – Major version of the learner

minor
int – Minor version of the learner

patch
int – Patch version of the learner

model_name
str – Name of the model

model_type
str – Type of the model, either regression or classification

_model
Any – The instantiated model, such as a Keras compiled model

_val_loss
dict – Validation loss of the model according to the metrics defined in its implementation

path
Path on disk of the model :returns: :rtype: str

sql
SQLized representation of model metadata

Returns The SQLAlchemy representation of the model

Return type *MLModel*

class `racket.`**KerasLearner**
Base class providing functionality for training & storing a model

build_model()
Abstract method. Must be overridden. Raises: `NotImplementedError` if called from base class

fit(*x*, *y*, **args*, ***kwargs*)
Abstract method. Must be overridden. Raises: `NotImplementedError` if called from base class

Parameters

- **x** (*array_like*) – a numpy array, or matrix that serves as input to the model. Must have matching dimensions to the model input specs
- **y** (*array_like*) – the targets for the input data
- **args** – Other parameters to be fed to the model
- **kwargs** – Other parameters to be fed to the model

historic_scores
Only available when model has been fit. Provides access to the latest validation scores

Returns Dictionary of metric scores `{metric:  score}`

Return type dict

model
returns: The compiled model :rtype: Sequential

scores(*x: Iterable*, *y: Iterable*) → object
Evaluate scores on a test set

Parameters

- **x** (*array_like*) – A numpy array, or matrix that serves as input to the model. Must have matching dimensions to the model input specs
- **y** (*array_like*) – the targets for the input data

Returns Dictionary of metric scores `{metric:  score}` evaluated on the test set

Return type dict

store(*autoload: bool = False*) → None
Stores the model in three different ways/patterns:

1. Keras serialization, that is a json + h5 object, from which it can be loaded into a TensorFlow session
2. TensorFlow protocol buffer + variables. That is the canonical TensorFlow way of storing models
3. Metadata, scores, and info about the model are stored in a relational database for tracking purposes

Returns

Return type None

tf_path
On disk path of the TensorFlow serialized model :returns: :rtype: str

class `racket.operations.load.`**ModelLoader**
This class provides the interface to load new models into TensorFlow Serving. This is implemented through a gRPC call to the TFS api which triggers it to look for directories matching the name of the model specified

classmethod load(*model_name: str*) → None
Load model

This will send the gRPC request. In particular, it will open a gRPC channel and communicate with the ReloadConfigRequest api to inform TFS of a change in configuration

Parameters **model_name** (*str*) – Name of the model, as specified in the instantiated Learner class

Returns

Return type None

class `racket.models.base.`**MLModel**(***kwargs*)
The SQL DeclarativeMeta model responsible for storing a model's metadata

Parameters

- **model_id** (*int*) – The model's unique identifier
- **model_name** (*str*) – Model name, usually defined with instantiating a Learner class
- **major** (*int*) – Major version of the learner
- **minor** (*int*) – Minor version of the learner
- **patch** (*int*) – Patch version of the learner
- **version_dir** (*str*) – Directory where the models will be stored inside TensorFlow serving and on-disk
- **created_at** (*dateteime.datetime*) – When the model was created
- **model_type** (*str*) – The model type usually either regression or classification

class `racket.models.base.`**MLModelInputs**(***kwargs*)

class racket.models.base.**ModelScores**(***kwargs*)
Scores of the model

Parameters

- **model_id** (*int*) – The model's unique identifier
- **scoring_fn** (*str*) – The name of the scoring function
- **score** (*float*) – The cross-validation score associated with the scoring function and the model id

class racket.models.channel.**Channel**
A gRPC channel implementation

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/carlomazzaferro/racket/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

racket could always use more documentation, whether as part of the official racket docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/carlomazzaferro/racket/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *racket* for local development.

1. Fork the *racket* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/racket.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   $ mkvirtualenv racket
   $ cd racket/
   $ python setup.py develop
   ```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

   ```
   $ python setup.py test
   $ tox
   ```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, unless it is a python compatibility request that targets a specific python release. Check https://travis-ci.org/carlomazzaferro/racket/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_racket
```

6.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis will then deploy to PyPI if tests pass.

CHAPTER 7

Credits

7.1 Development Lead

- Carlo Mazzaferro <carlo.mazzaferro@gmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

8.1 0.1.0 (2018-11-02)

- First release on PyPI.

8.2 0.2.0 (2018-11-13)

- Major feature implementation and documentation
- Static typing
- Testing - 78% coverage

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index