DEEPaaS Documentation

Release 2.0.0

DEEP-Hybrid-DataCloud consortium

Jun 08, 2022

CONTENTS

Release v2.0.0. (*Installation*)

DEEP as a Service (DEEPaaS) is a REST API that is focused on providing access to machine learning models. By using DEEPaaS users can easily run a REST API in fron of their model, thus accesing its functionality via HTTP calls.

CHAPTER

ONE

COMPATIBILITY

The DEEPaaS API works with Python 3.5.3 or higher.

1.1 Upgrading from V1 of the API

The DEEPaaS API provides two different API versions: V1 and V2. V1 was the initial API version, supported in releases under `1.0.0` and compatible with Python 2.7+ and Python 3. Starting with the `1.0.0` release V2 is the only version supported and requires Python 3.5.3 or higher. Please, read carefully the *Upgrade notes* before upgrading to a newer version.

CHAPTER

TWO

INSTALLATION AND UPGRADE DOCUMENTATION

2.1 Installation, upgrade and configuration

2.1.1 Installation

The recommended way to install the DEEPaaS API is with `pip`:

```
pip install deepass
```

This way you will ensure that the stable version is fetched from pip, rather than development or unstable version

Installing the development version

The development version of DEEPaas can be installed from the `master` brach of the GitHub DEEPaaS repository and can be installed as follows (note the `-e` switch to install it in editable or "develop mode"):

```
git clone https://github.com/indigo-dc/DEEPaaS
cd DEEPaaS
pip install -e
```

2.1.2 Upgrade notes

Upgrading to version 2.0.0

The release `2.0.0` drops the support for the V1 version of the API. Before upgrading your code and deploying into production please ensure that you are no longer using the V1 version of the API.

Upgrading to version 1.0.0

The release `1.0.0` of the DEEPaaS API implements several backwards incompatible changes when compared with the previous releases in the `0.X.X` series. Before upgrading your code and deploying into production, read the following, as changes are needed in your model. Please go through the following checklist in order.

- **Python version**

  The new version of DEEPaaS uses `aiohttp` which requires at least Python 3.5.3. So please update you module accordingly if needed.

- **Migrate new namespace entry point.**

  Previous code relied on a top-level entry point named `deepaas.model`, where we searched for the required functions or methods to be invoked for each API action.

  Now the namespace has changed to `deepaas.v2.model`.

  Therefore, assuming that your code for V1 of the API was under the `my_model.api` and you defined your the entry point as follows:

```
[entry_points]

deepaas.model =
  my_model = my_model.api
```

  You should migrate to the following:

```
[entry_points]

deepaas.v2.model =
    my_model = my_model.api
```

  Note: If you do not change the namespace, we will try to load the old entrypoint. However, this is deprecated and you should upgrade as soon as possible.

- **Migrate from returning dictionaries to use an argument parser to define train and predict arguments.**

  Previous code relied on returning arbitrary dictionaries that were used to generate the arguments for each of the API endpoints. This is not anymore supported and you should return a `webargs` field dictionary (check here for more information. This is still done by defining the `get_predict_args` and `get_train_args` functions. These functions must receive no arguments and they should return a dictionary as follows:

```
from webargs import fields

(...)

def get_predict_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

  Note: If you do still follow the old way of returning the arguments we will try to load the arguments using the old `get_*_args` functions. However, this is DEPRECATED and you should migrate to using the argument parser as soon as possible. All the arguments will be converted to Strings, therefore you will loose any type checking, etc.

- **Explicitly define your input arguments.**

The previous version of the API defined two arguments for inference: `data` and `urls`. This is not anymore true, and you must define your own input arguments. To replicate the response of v1 you have to define:

```
from webargs import fields

(...)

def get_predict_args():
   return {
      'files': fields.Field(
          required=False,
          missing=None,
          type="file",
          data_key="data",
          location="form",
          description="Select the image you want to classify."),

      'urls': fields.Url(
          required=False,
          missing=None,
          description="Select an URL of the image you want to classify.")
           }
```

Then, you will get your input data in the `data` and `urls` keyword arguments in your application.

Note: For the moment, in contrast with v1, only one url field at the same time is enabled, although multi-url (along with multi-files) support is coming soon.

- **Define your responses for the prediction.**

  Now, unless you explicitly define your application response schema, whatever you return will be converted into a string and wrapped in the following response:

```
{
   "status": "OK",
   "predictions": "<model response as string>"
}
```

- **Change in the ``predict`` function name.**

  The `predict_url` and `predict_data` functions have been merged into a single `predict` function. In addition, arguments are now passed as unpacked keyword arguments, not anymore as a dictionary. So if you want to upgrade to v2 with minimal code changes, you just have to add the following function to your .py file:

```
def predict(**args):

    if (not any([args['urls'], args['files']]) or
            all([args['urls'], args['files']])):
        raise Exception("You must provide either 'url' or 'data' in the payload")

    if args['files']:
        args['files'] = [args['files']]  # patch until list is available
        return predict_data(args)
    elif args['urls']:
```

(continues on next page)

(continued from previous page)

```
        args['urls'] = [args['urls']]  # patch until list is available
        return predict_url(args)
```

- **Changes in the data response**

  The returned object in `args['files']` is no longer a `werkzeug.FileStorage` but a `deepaas.model.v2.wrapper.UploadedFile` which has attributes like `name` (name of the argument where this file is being sent), `filename` (complete file path to the temporary file in the filesystem) and `content_type` (content-type of the uploaded file).

  The main difference is that now you should read the bytes using `open(f.filename, 'rb').read()` instead of `f.read()`.

- **Catch error function**

  By default Exceptions raised in the PREDICT method from the application side will be rendered as `500 - HTTPInternalServerError` with the message `Server got itself in trouble`. If you want to render some Exceptions with custom status codes and custom messages (see the `reason` arg) you have to raise an aiohttp web exception. For example:

```
from aiohttp.web import HTTPBadRequest

try:
    f()
except Exception as e:
    raise HTTPBadRequest(reason=e)
```

  And if you want to wrap a whole function so that any error it raises is passed as a certain HTTP error:

```
def catch_error(f):
    def wrap(*args, **kwargs):
        try:
            return f(*args, **kwargs)
        except Exception as e:
            raise HTTPBadRequest(reason=e)
    return wrap


@catch_error
def f():
    ...
```

  Note: Python Exceptions from the application side for the TRAIN method will be correctly rendered by DEEPaaS so there is no need to wrap the train function with the catch_error decorator.

- **API url**

  Now the API functions are accessed under http://api_url/docs (eg. http://0.0.0.0:5000/docs)

2.1.3 Configuration Guide

The configuration for DEEPaaS is handled by the `deepaas.conf` configuration file, described below.

- *Config Reference*: A complete reference of all configuration options available in the `deepaas.conf` file.
- *Sample Config File*: A sample config file with inline documentation.

2.1.4 DEEPaaS Release Notes

Current Release Notes

2.0.0

Prelude

Fallback loading of arguments (from V1 style) is now removed.

This new release removes support for the V1 API (marked as deprecated in version 1.0.0).

Upgrade Notes

- The old way of returning the arguments, to load the arguments using the old `get_*_args` functions is now removed. If you were still relying on them (check the logs) you must migrate to using the argument parser as soon as possible.
- V1 API is now removed, therefore all modules using the `deepaas.model` entrypoint will stop working. The `--enable-v1` configuration option is now removed.

Other Notes

- V1 support is now removed.

1.3.0

New Features

- Include `deepaas-cli` a new command line tool that provides the same functionality as the API through the commandline. This is useful for execution of batch tasks in a HPC/HTC or batch system cluster.

1.2.1

Bug Fixes

- Fix [#83](https://github.com/indigo-dc/DEEPaaS/issues/87) out out memory errors due to the usage of two different executor pools.

1.2.0

New Features

- Include a new command line tool to execute inference and prediction calls from the shell, whitout spawning a server and making cURL requests to it. This is useful for execution of batch tasks in a HPC/HTC or batch system cluster.

Bug Fixes

- This new release fixes an error introduced by *webargs* in its 6.0.0 version. Since we cannot yet fix it (*aiohttp-webargs* needs to be updated as well) we pin webargs to the 5.X.X versions https://github.com/indigo-dc/DEEPaaS/issues/82
- Set *debug* in OpenWhisk mode as configured by the user.

CHAPTER

THREE

USER DOCUMENTATION

If you want to try the DEEPaaS API, or if you want to integrate a machine learning, neural network or deep learning model with it, this documentation is what you are looking for.

3.1 User and developer documentation

3.1.1 Quickstart

The best way to quickly try the DEEPaaS API is by issuing the following command (no installation required):

```
make run
```

This command will create a Python virtualenv (in the `virtualenv` directory) with DEEPaaS along all its dependencies. Then it will run the DEEPaaS REST API, listening on `http://127.0.0.1:5000`. If you browse to that URL you will get the Swagger UI documentation page.

If you want to run the code in the editable or develop mode (i.e. the same environment that you would get with `pip install -e`), you can issue the following command before:

```
make develop
```

3.1.2 Integrating a model into the V2 API (CURRENT)

Important: V2 of the API (starting on release `1.0.0`) is the default, supported version. It is backwards incompatible with the V1 version.

Note: The current version of the DEEPaaS API is V2. The first release supporting this API version was `1.0.0`. Please do not be confused with the `deepaas` *release* (i.e. `0.5.2`, `1.0.0`) and the DEEPaaS API version (V1 or V2).

Defining what to load

The DEEPaaS API uses Python's Setuptools entry points that are dynamically loaded to offer the model functionality through the API. This allows you to offer several models using a single DEEPaaS instance, by defining different entry points for the different models.

When the DEEPaaS API is spawned it will look for the `deepaas.v2.model` entrypoint namespace, loading and adding the names found into the API namespace. In order to define your entry points, your module should leverage setuptools and be ready to be installed in the system. Then, in order to define your entry points, you should add the following to your `setup.cfg` configuration file:

```
[entry_points]

deepaas.v2.model =
    my_model = package_name.module
```

This will define an entry point in the `deepaas.v2.model` namespace, called `my_model`. All the required functionality will be fetched from the `package_name.module` module. This means that `module` should provide the model-api as described below.

If you provide a class with the required functionality, the entry point will be defined as follows:

```
[entry_points]

deepaas.v2.model =
    my_model = package_name.module:Class
```

Again, this will define an entry point in the `deepaas.v2.model` namespace, called `my_model`. All the required functionality will be fetched from the `package_name.module.Class` class, meaning that an object of `Class` will be created and used as entry point. This also means that `Class` objects should provide the model-api as described below.

Entry point (model) API

Regardless on the way you implement your entry point (i.e. as a module or as an object), you should expose the following functions or methods:

Defining model metadata

Your model entry point must implement a `get_medatata` function that will return some basic metadata information about your model, as follows:

`get_metadata`(*self*)

Return metadata from the exposed model.

The metadata that is expected should follow the schema that is shown below. This basically means that you should return a dictionary with the following aspect:

```
{
    "author": "Author name",
    "description": "Model description",
    "license": "Model's license",
    "url": "URL for the model (e.g. GitHub repository)",
    "version": "Model version",
}
```

The only fields that are mandatory are 'description' and 'name'.

The schema that we are following is the following:

```
{
    "id": =  fields.Str(required=True,
                        description='Model identifier'),
    "name": fields.Str(required=True,
                       description='Model name'),
    "description": fields.Str(required=True,
                              description='Model description'),
    "license": fields.Str(required=False,
                          description='Model license'),
    "author": fields.Str(required=False,
                         description='Model author'),
    "version": fields.Str(required=False,
                          description='Model version'),
    "url": fields.Str(required=False,
                      description='Model url'),
    "links": fields.List(
        fields.Nested(
            {
                "rel": fields.Str(required=True),
                "href": fields.Url(required=True),
            }
        )
    )
}
```

Returns dictionary containing the model's metadata.

Warming a model

You can initialize your model before any prediction or train is done by defining a `warm` function. This function receives no arguments and returns no result, but it will be call before the API is spawned.

You can use it to implement any loading or initialization that your model may use. This way, your model will be ready whenever a first prediction is done, reducint the waiting time.

warm(*self*)

Warm (initialize, load) the model.

This is called when the model is loaded, before the API is spawned.

If implemented, it should prepare the model for execution. This is useful for loading it into memory, perform any kind of preliminary checks, etc.

Training

Regarding training there are two functions to be defined. First of all, you can specify the training arguments to be defined (and published through the API) with the `get_train_args` function, as follows:

`get_train_args`(*self*)

Return the arguments that are needed to train the application.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_train_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

Then, you must implement the training function (named `train`) that will receive the defined arguments as keyword arguments:

`train`(*self*, ***kwargs*)

Perform a training.

Parameters **`kwargs`** – The keyword arguments that the predict method accepts must be defined by the `get_train_args()` method so the API is able to pass them down. Usually you would populate these with all the training hyper-parameters

Returns TBD

Prediction and inference

For prediction, there are different functions to be implemented. First of all, as for the training, you can specify the prediction arguments to be defined, (and published through the API) with the `get_predict_args` as follows:

`get_predict_args`(*self*)

Return the arguments that are needed to perform a prediction.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_predict_args():
    return {
        "arg1": fields.Str(
```

(continues on next page)

(continued from previous page)

```
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

Do not forget to add an input argument to hold your data. If you want to upload files for inference to the API, you should use a `webargs.fields.Field` field created as follows:

```
def get_predict_args():
    return {
        "data": fields.Field(
            description="Data file to perform inference on.",
            required=False,
            missing=None,
            type="file",
            location="form")
     }
```

You can also predict data stored in an URL by using:

```
def get_predict_args():
    return {
        "url": fields.Url(
            description="Url of data to perform inference on.",
            required=False,
            missing=None)
     }
```

Important: do not forget to add the `location="form"` and `type="file"` to the argument definition, otherwise it will not work as expected.

Once defined, you will receive an object of the class described below for each of the file arguments you declare. You can open and read the file stored in the `filename` attribute.

class UploadedFile(*name*, *filename*, *content_type*, *original_filename*)
Class to hold uploaded field metadata when passed to model's methods

name
Name of the argument where this file is being sent.

filename
Complete file path to the temporary file in the filesystem,

content_type
Content-type of the uploaded file

original_filename
Filename of the original file being uploaded.

Then you should define the `predict` function as indicated below. You will receive all the arguments that have been parsed as keyword arguments:

predict(*self*, ***kwargs*)

Prediction from incoming keyword arguments.

Parameters kwargs – The keyword arguments that the predict method accepts must be defined by the `get_predict_args()` method so the API is able to pass them down.

Returns The response must be a str or a dict.

By default, the return values from these two functions will be casted into a string, and will be returned in the following JSON response:

```
{
   "status": "OK",
   "predictions": "<model response as string>"
}
```

However, it is recommended that you specify a custom response schema. This way the API exposed will be richer and it will be easier for developers to build applications against your API, as they will be able to discover the response schemas from your endpoints.

In order to define a custom response, the `response` attribute is used:

Returning different content types

Sometimes it is useful to return something different than a JSON file. For such cases, you can define an additional argument `accept` defining the content types that you are able to return as follows:

```
def get_predict_args():
    return {
        'accept': fields.Str(description="Media type(s) that is/are acceptable for the␣
↪response.",
                              missing='application/zip',
                              validate=validate.OneOf(['application/zip', 'image/png',
↪'application/json']))
     }
```

Find here a comprehensive list of possible content types. Then the predict function will have to return the raw bytes of a file according to the user selection. For example:

```
def predict(**args):
    # Run your prediction

    # Return file according to user selection
    if args['accept'] == 'image/png':
        return open(img_path, 'rb')

    elif args['accept'] == 'application/json':
        return {'some': 'json'}

    elif args['accept'] == 'application/zip':
        return open(zip_path, 'rb')
```

If you want to return several content types at the same time (let's say a JSON and an image), the easiest way it to return a zip file with all the files.

Using classes

Apart from using a module, you can base your entrypoints on classes. If you want to do so, you may find useful to inhering from the `deepaas.model.v2.base.BaseModel` abstract class:

class BaseModel

Base class for all models to be used with DEEPaaS.

Note that it is not needed for DEEPaaS to inherit from this abstract base class in order to expose the model functionality, but the entrypoint that is configured should expose the same API.

abstract get_metadata()

Return metadata from the exposed model.

The metadata that is expected should follow the schema that is shown below. This basically means that you should return a dictionary with the following aspect:

```
{
    "author": "Author name",
    "description": "Model description",
    "license": "Model's license",
    "url": "URL for the model (e.g. GitHub repository)",
    "version": "Model version",
}
```

The only fields that are mandatory are 'description' and 'name'.

The schema that we are following is the following:

```
{
    "id": =  fields.Str(required=True,
                        description='Model identifier'),
    "name": fields.Str(required=True,
                       description='Model name'),
    "description": fields.Str(required=True,
                              description='Model description'),
    "license": fields.Str(required=False,
                          description='Model license'),
    "author": fields.Str(required=False,
                         description='Model author'),
    "version": fields.Str(required=False,
                          description='Model version'),
    "url": fields.Str(required=False,
                      description='Model url'),
    "links": fields.List(
        fields.Nested(
            {
                "rel": fields.Str(required=True),
                "href": fields.Url(required=True),
            }
        )
    )
}
```

Returns dictionary containing the model's metadata.

abstract get_predict_args()

Return the arguments that are needed to perform a prediction.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_predict_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

abstract get_train_args()

Return the arguments that are needed to train the application.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_train_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

abstract predict(***kwargs*)

Prediction from incoming keyword arguments.

Parameters **kwargs** – The keyword arguments that the predict method accepts must be defined by the `get_predict_args()` method so the API is able to pass them down.

Returns The response must be a str or a dict.

schema = None

Must contain a valid schema for the model's predictions or None.

A valid schema is either a `marshmallow.Schema` subclass or a dictionary schema that can be converted into a schema.

In order to provide a consistent API specification we use this attribute to define the schema that all the prediction responses will follow, therefore: - If this attribute is set we will validate them against it. - If it is not set (i.e. `schema = None`), the model's response will

be converted into a string and the response will have the following form:

```
{
    "status": "OK",
    "predictions": "<model response as string>"
}
```

As previously stated, there are two ways of defining an schema here. If our response have the following form:

```
{
    "status": "OK",
    "predictions": [
        {
            "label": "foo",
            "probability": 1.0,
        },
        {
            "label": "bar",
            "probability": 0.5,
        },
    ]
}
```

We should define or schema as schema as follows:

- Using a schema dictionary. This is the most straightforward way. In order to do so, you must use the `marshmallow` Python module, as follows:

```
from marshmallow import fields

schema = {
    "status": fields.Str(
                description="Model predictions",
                required=True
    ),
    "predictions": fields.List(
        fields.Nested(
            {
                "label": fields.Str(required=True),
                "probability": fields.Float(required=True),
            },
        )
    )
}
```

- Using a `marshmallow.Schema` subclass. Note that the schema *must* be the class that you have created, not an object:

```python
import marshmallow
from marshmallow import fields

class Prediction(marshmallow.Schema):
    label = fields.Str(required=True)
    probability = fields.Float(required=True)

class Response(marshmallow.Schema):
    status = fields.Str(
        description="Model predictions",
        required=True
    )
    predictions = fields.List(fields.Nested(Prediction))

schema = Response
```

abstract train(***kwargs*)

Perform a training.

Parameters kwargs – The keyword arguments that the predict method accepts must be defined by the `get_train_args()` method so the API is able to pass them down. Usually you would populate these with all the training hyper-parameters

Returns TBD

abstract warm()

Warm (initialize, load) the model.

This is called when the model is loaded, before the API is spawned.

If implemented, it should prepare the model for execution. This is useful for loading it into memory, perform any kind of preliminary checks, etc.

Warning: The API uses `multiprocessing` for handling tasks. Therefore if you use decorators around your methods, please follow best practices and use functools.wraps so that the methods are still pickable. Beware also of using global variables that might not be shared between processes.

3.1.3 DEEPaaS API as an OpenWhisk action

DEEPaaS API can be executed inside a Docker container as an OpenWhisk Docker action. In your Dockerfile, you have to ensure that you execute `deepaas-run` with the `--openwhisk-detect` switch, as follows:

```
(...)
CMD ["sh", "-c", "deepaas-run --openwhisk --listen-ip 0.0.0.0"]
```

For a complete example, check the *DEEP OC Generic Container <https://github.com/deephdc/DEEP-OC-generic-container>* or the `Dockerfile` that is included within the DEEPaaS API repository.

With that Dockerfile you can build your docker container and create the corresponding OpenWhisk action:

```
docker build -t foobar/container .
wsk action create action-name --docker foobar/container --web true
```

3.2 API reference

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

3.2.1 Internal API

V2 Models

Base Model

class BaseModel

Base class for all models to be used with DEEPaaS.

Note that it is not needed for DEEPaaS to inherit from this abstract base class in order to expose the model functionality, but the entrypoint that is configured should expose the same API.

abstract get_metadata()

Return metadata from the exposed model.

The metadata that is expected should follow the schema that is shown below. This basically means that you should return a dictionary with the following aspect:

```
{
    "author": "Author name",
    "description": "Model description",
    "license": "Model's license",
    "url": "URL for the model (e.g. GitHub repository)",
    "version": "Model version",
}
```

The only fields that are mandatory are 'description' and 'name'.

The schema that we are following is the following:

```
{
    "id": =  fields.Str(required=True,
                        description='Model identifier'),
    "name": fields.Str(required=True,
                       description='Model name'),
    "description": fields.Str(required=True,
                              description='Model description'),
    "license": fields.Str(required=False,
                          description='Model license'),
    "author": fields.Str(required=False,
                         description='Model author'),
    "version": fields.Str(required=False,
                          description='Model version'),
    "url": fields.Str(required=False,
                      description='Model url'),
    "links": fields.List(
        fields.Nested(
            {
                "rel": fields.Str(required=True),
                "href": fields.Url(required=True),
```

(continues on next page)

(continued from previous page)

```
                }
            )
        )
}
```

Returns dictionary containing the model's metadata.

abstract get_predict_args()

Return the arguments that are needed to perform a prediction.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_predict_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

abstract get_train_args()

Return the arguments that are needed to train the application.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_train_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

abstract predict(*kwargs*)**

Prediction from incoming keyword arguments.

Parameters **kwargs** – The keyword arguments that the predict method accepts must be defined by the `get_predict_args()` method so the API is able to pass them down.

Returns The response must be a str or a dict.

schema = None

Must contain a valid schema for the model's predictions or None.

A valid schema is either a `marshmallow.Schema` subclass or a dictionary schema that can be converted into a schema.

In order to provide a consistent API specification we use this attribute to define the schema that all the prediction responses will follow, therefore: - If this attribute is set we will validate them against it. - If it is not set (i.e. `schema = None`), the model's response will

be converted into a string and the response will have the following form:

```
{
    "status": "OK",
    "predictions": "<model response as string>"
}
```

As previously stated, there are two ways of defining an schema here. If our response have the following form:

```
{
    "status": "OK",
    "predictions": [
        {
            "label": "foo",
            "probability": 1.0,
        },
        {
            "label": "bar",
            "probability": 0.5,
        },
    ]
}
```

We should define or schema as schema as follows:

- Using a schema dictionary. This is the most straightforward way. In order to do so, you must use the `marshmallow` Python module, as follows:

```
from marshmallow import fields

schema = {
    "status": fields.Str(
                description="Model predictions",
                required=True
    ),
    "predictions": fields.List(
        fields.Nested(
            {
                "label": fields.Str(required=True),
                "probability": fields.Float(required=True),
            },
```

(continues on next page)

(continued from previous page)

```
        )
    )
}
```

- Using a `marshmallow.Schema` subclass. Note that the schema *must* be the class that you have created, not an object:

```
import marshmallow
from marshmallow import fields

class Prediction(marshmallow.Schema):
    label = fields.Str(required=True)
    probability = fields.Float(required=True)

class Response(marshmallow.Schema):
    status = fields.Str(
        description="Model predictions",
        required=True
    )
    predictions = fields.List(fields.Nested(Prediction))

schema = Response
```

abstract train(***kwargs*)

Perform a training.

Parameters kwargs – The keyword arguments that the predict method accepts must be defined by the `get_train_args()` method so the API is able to pass them down. Usually you would populate these with all the training hyper-parameters

Returns TBD

abstract warm()

Warm (initialize, load) the model.

This is called when the model is loaded, before the API is spawned.

If implemented, it should prepare the model for execution. This is useful for loading it into memory, perform any kind of preliminary checks, etc.

Model wrapper

V2 test model

class TestModel

Dummy model implementing minimal functionality.

This is a simple class that mimics the behaviour of a module, just for showing how the whole DEEPaaS works and documentation purposes.

get_metadata()

Return metadata from the exposed model.

The metadata that is expected should follow the schema that is shown below. This basically means that you should return a dictionary with the following aspect:

```
{
    "author": "Author name",
    "description": "Model description",
    "license": "Model's license",
    "url": "URL for the model (e.g. GitHub repository)",
    "version": "Model version",
}
```

The only fields that are mandatory are 'description' and 'name'.

The schema that we are following is the following:

```
{
    "id": =  fields.Str(required=True,
                        description='Model identifier'),
    "name": fields.Str(required=True,
                       description='Model name'),
    "description": fields.Str(required=True,
                              description='Model description'),
    "license": fields.Str(required=False,
                          description='Model license'),
    "author": fields.Str(required=False,
                         description='Model author'),
    "version": fields.Str(required=False,
                          description='Model version'),
    "url": fields.Str(required=False,
                      description='Model url'),
    "links": fields.List(
        fields.Nested(
            {
                "rel": fields.Str(required=True),
                "href": fields.Url(required=True),
            }
        )
    )
}
```

Returns dictionary containing the model's metadata.

get_predict_args()

Return the arguments that are needed to perform a prediction.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_predict_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
```

(continues on next page)

(continued from previous page)

```
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

`get_train_args()`

Return the arguments that are needed to train the application.

This function should return a dictionary of `webargs` fields (check here for more information). For example:

```
from webargs import fields

(...)

def get_train_args():
    return {
        "arg1": fields.Str(
            required=False,  # force the user to define the value
            missing="foo",  # default value to use
            enum=["choice1", "choice2"],  # list of choices
            description="Argument one"  # help string
        ),
    }
```

Return dict A dictionary of `webargs` fields containing the application required arguments.

`predict`(***kwargs*)

Prediction from incoming keyword arguments.

Parameters **`kwargs`** – The keyword arguments that the predict method accepts must be defined by the `get_predict_args()` method so the API is able to pass them down.

Returns The response must be a str or a dict.

`train`(**args*, ***kwargs*)

Perform a training.

Parameters **`kwargs`** – The keyword arguments that the predict method accepts must be defined by the `get_train_args()` method so the API is able to pass them down. Usually you would populate these with all the training hyper-parameters

Returns TBD

`warm()`

Warm (initialize, load) the model.

This is called when the model is loaded, before the API is spawned.

If implemented, it should prepare the model for execution. This is useful for loading it into memory, perform any kind of preliminary checks, etc.

CHAPTER FOUR

ADDITIONAL NOTES

4.1 How to contribute to DEEPaaS API

# Contributing guidelines

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

## Types of Contributions

You can contribute in many ways:

### Report Bugs

Report bugs at https://github.com/IFCA/deepaas/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.
- If you can, provide detailed steps to reproduce the bug.
- If you don't have steps to reproduce the bug, just note your observations in as much detail as you can. Questions to start a discussion about the issue are welcome.

### Fix Bugs

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

### Implement Features

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

Please do not combine multiple feature enhancements into a single pull request.

Note: this project is very conservative, so new features that aren't tagged with "please-help" might not get into core. We're trying to keep the code base small, extensible, and streamlined. Whenever possible, it's best to try and implement feature ideas as separate projects outside of the core codebase.

### Write Documentation

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

If you want to review your changes on the documentation locally, you can do:

```
pip install -r docs/requirements.txt
make servedocs
```

This will compile the documentation, open it in your browser and start watching the files for changes, recompiling as you save.

### Submit Feedback

The best way to send feedback is to file an issue at the follwing URL:

> https://github.com/IFCA/deepaas/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 :)

## Setting Up the Code for Local Development

Here's how to set up *deepaas* for local development.

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

   ```
   $ git clone git@github.com:your_name_here/deepaas.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 deepaas
   $ cd deepaas/
   $ 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 the tests and the style checks (pep8, flake8 and https://docs.openstack.org/hacking/latest/):

   > $ pip install tox $ tox

Please note that tox runs the style tests automatically, since we have a test environment for it (named pep8).

If you feel like running only the pep8 environment, please use the following command:

```
$ tox -e pep8
```

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. Check that the test coverage hasn't dropped:

   ```
   $ tox -e cover
   ```

8. Submit a pull request through the GitHub website.

## Contributor Guidelines

### 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 2.7, 3.4, 3.6 on Travis CI.
4. Check https://travis-ci.org/IFCA/deepaas/pull_requests to ensure the tests pass for all supported Python versions and platforms.

### Coding Standards

- PEP8
- We follow the OpenStack Style Guidelines: https://docs.openstack.org/hacking/latest/user/hacking.html#styleguide
- Write new code in Python 3.

## Testing with tox

Tox uses *py.test* under the hood, hence it supports the same syntax for selecting tests.

For further information please consult the **`pytest usage docs`_**.

To run a particular test class with tox:

```
$ tox -e py '-k TestCasoManager'
```

To run some tests with names matching a string expression:

```
$ tox -e py '-k generate'
```

Will run all tests matching "generate", test_generate_files for example.

To run just one method:

```
$ tox -e py '-k "TestCasoManager and test_required_fields"'
```

To run all tests using various versions of python in virtualenvs defined in tox.ini, just run tox.:

```
$ tox
```

This configuration file setup the pytest-cov plugin and it is an additional dependency. It generate a coverage report after the tests.

It is possible to tests with some versions of python, to do this the command is:

$ tox -e py27,py34

Will run py.test with the python2.7, python3.4 and pypy interpreters, for example.

# Contributor Covenant Code of Conduct

## Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

## Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

## Our Responsibilities

Project maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

## Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

## Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at [INSERT EMAIL ADDRESS]. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The project team is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

Project maintainers who do not follow or enforce the Code of Conduct in good faith may face temporary or permanent repercussions as determined by other members of the project's leadership.

## Attribution

This Code of Conduct is adapted from the [Contributor Covenant][homepage], version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

[homepage]: https://www.contributor-covenant.org

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

5.1 Configuration Options

The following is an overview of all available configuration options in Nova. For a sample configuration file, refer to *Sample Configuration File*.

5.1.1 DEFAULT

`listen_ip`

Type string

Default `127.0.0.1`

IP address on which the DEEPaaS API will listen.

The DEEPaaS API service listens on this IP address for incoming requests.

`listen_port`

Type port number

Default `5000`

Minimum Value 0

Maximum Value 65535

Port on which the DEEPaaS API will listen.

The DEEPaaS API service listens on this port number for incoming requests.

`openwhisk_detect`

Type boolean

Default `False`

Run as an OpenWhisk action.

If this option is set to True DEEPaaS will check if the __OW_API_HOST environment variable is set. If it is set, it will run an OpenWhisk Docker action listener rather than the DEEPaaS API. If it is not set, it will run a DEEPaaS in normal mode.

If you specify this option, the value of 'listen-ip' will be used, but the port will is hardcoded to 8080 (as OpenWhisk goes to port 8080). Note that if you are running inside a container, the most sensible option is to set listen-ip to 0.0.0.0

`debug_endpoint`

Type boolean

Default `false`

Enable debug endpoint. If set we will provide all the information that you print to the standard output and error (i.e. stdout and stderr) through the "/debug" endpoint. Default is to not provide this information. This will not provide logging information about the API itself.

`workers`

Type integer

Default `1`

Specify the number of workers to spawn. If using a CPU you probably want to increase this number, if using a GPU probably you want to leave it to 1. (defaults to 1)

`client_max_size`

Type integer

Default `0`

Minimum Value 0

Client's maximum size in a request, in bytes. If a POST request exceeds this value, it raises an HTTPRequestEntityTooLarge exception. If set to 0, no file size limit will be enforced.

`warm`

Type boolean

Default `True`

Pre-warm the modules (eg. load models, do preliminary checks, etc). You might want to disable this option if DEEPaaS is loading more than one module because you risk getting out of memory errors.

5.1.2 Logging options

DEFAULT

`debug`

Type boolean

Default `False`

Mutable This option can be changed without restarting.

If set to true, the logging level will be set to DEBUG instead of the default INFO level.

`log_config_append`

Type string

Default `<None>`

Mutable This option can be changed without restarting.

The name of a logging configuration file. This file is appended to any existing logging configuration files. For details about logging configuration files, see the Python logging module documentation. Note that when logging configuration files are used then all logging configuration is set in the configuration file and other logging configuration options are ignored (for example, log-date-format).

Table 1: Deprecated Variations

Group	Name
DEFAULT	log-config
DEFAULT	log_config

`log_date_format`

Type string

Default `%Y-%m-%d %H:%M:%S`

Defines the format string for %(asctime)s in log records. Default: the value above . This option is ignored if log_config_append is set.

`log_file`

Type string

Default `<None>`

(Optional) Name of log file to send logging output to. If no default is set, logging will go to stderr as defined by use_stderr. This option is ignored if log_config_append is set.

Table 2: Deprecated Variations

Group	Name
DEFAULT	logfile

`log_dir`

Type string

Default `<None>`

(Optional) The base directory used for relative log_file paths. This option is ignored if log_config_append is set.

Table 3: Deprecated Variations

Group	Name
DEFAULT	logdir

`watch_log_file`

Type boolean

Default `False`

Uses logging handler designed to watch file system. When log file is moved or removed this handler will open a new log file with specified path instantaneously. It makes sense only if log_file option is specified and Linux platform is used. This option is ignored if log_config_append is set.

`use_syslog`

Type boolean

Default `False`

Use syslog for logging. Existing syslog format is DEPRECATED and will be changed later to honor RFC5424. This option is ignored if log_config_append is set.

`use_journal`

Type boolean

Default `False`

Enable journald for logging. If running in a systemd environment you may wish to enable journal support. Doing so will use the journal native protocol which includes structured metadata in addition to log messages.This option is ignored if log_config_append is set.

`syslog_log_facility`

Type string

Default `LOG_USER`

Syslog facility to receive log lines. This option is ignored if log_config_append is set.

`use_json`

Type boolean

Default `False`

Use JSON formatting for logging. This option is ignored if log_config_append is set.

`use_stderr`

Type boolean

Default `False`

Log output to standard error. This option is ignored if log_config_append is set.

`use_eventlog`

Type boolean

Default `False`

Log output to Windows Event Log.

`log_rotate_interval`

Type integer

Default `1`

The amount of time before the log files are rotated. This option is ignored unless log_rotation_type is set to "interval".

`log_rotate_interval_type`

Type string

Default `days`

Valid Values Seconds, Minutes, Hours, Days, Weekday, Midnight

Rotation interval type. The time of the last file change (or the time when the service was started) is used when scheduling the next rotation.

`max_logfile_count`

Type integer

Default `30`

Maximum number of rotated log files.

`max_logfile_size_mb`

Type integer

Default `200`

Log file maximum size in MB. This option is ignored if "log_rotation_type" is not set to "size".

`log_rotation_type`

Type string

Default `none`

Valid Values interval, size, none

Log rotation type.

Possible values

interval Rotate logs at predefined time intervals.

size Rotate logs once they reach a predefined size.

none Do not rotate log files.

`logging_context_format_string`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s [%(global_request_id)s %(request_id)s %(user_identity)s] %(instance)s%(message)s`

Format string to use for log messages with context. Used by oslo_log.formatters.ContextFormatter

`logging_default_format_string`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d %(levelname)s %(name)s [-] %(instance)s%(message)s`

Format string to use for log messages when context is undefined. Used by oslo_log.formatters.ContextFormatter

`logging_debug_format_suffix`

Type string

Default `%(funcName)s %(pathname)s:%(lineno)d`

Additional data to append to log message when logging level for the message is DEBUG. Used by oslo_log.formatters.ContextFormatter

`logging_exception_prefix`

Type string

Default `%(asctime)s.%(msecs)03d %(process)d ERROR %(name)s %(instance)s`

Prefix each line of exception output with this format. Used by oslo_log.formatters.ContextFormatter `logging_user_identity_format`

Type string

Default `%(user)s %(project)s %(domain)s %(system_scope)s %(user_domain)s %(project_domain)s`

Defines the format string for %(user_identity)s that is used in logging_context_format_string. Used by oslo_log.formatters.ContextFormatter

`default_log_levels`

Type list

Default `['amqp=WARN', 'amqplib=WARN', 'boto=WARN', 'qpid=WARN', 'sqlalchemy=WARN', 'suds=INFO', 'oslo.messaging=INFO', 'oslo_messaging=INFO', 'iso8601=WARN', 'requests.packages.urllib3.connectionpool=WARN', 'urllib3.connectionpool=WARN', 'websocket=WARN', 'requests.packages.urllib3.util.retry=WARN', 'urllib3.util.retry=WARN', 'keystonemiddleware=WARN', 'routes.middleware=WARN', 'stevedore=WARN', 'taskflow=WARN', 'keystoneauth=WARN', 'oslo.cache=INFO', 'oslo_policy=INFO', 'dogpile.core.dogpile=INFO']`

List of package logging levels in logger=LEVEL pairs. This option is ignored if log_config_append is set.

`publish_errors`

Type boolean

Default `False`

Enables or disables publication of error events.

`instance_format`

Type string

Default `"[instance:  %(uuid)s] "`

The format for an instance that is passed with the log message.

`instance_uuid_format`

Type string

Default `"[instance:  %(uuid)s] "`

The format for an instance UUID that is passed with the log message.

`rate_limit_interval`

Type integer

Default `0`

Interval, number of seconds, of log rate limiting.

`rate_limit_burst`

Type integer

Default `0`

Maximum number of logged messages per rate_limit_interval.

`rate_limit_except_level`

Type string

Default `CRITICAL`

Log level name used by rate limiting: CRITICAL, ERROR, INFO, WARNING, DEBUG or empty string. Logs with level greater or equal to rate_limit_except_level are not filtered. An empty string means that all levels are filtered.

`fatal_deprecations`

Type boolean

Default `False`

Enables or disables fatal status of deprecations.

5.2 Sample Configuration File

The following is a sample DEEPaaS configuration for adaptation and use. For a detailed overview of all available configuration options, refer to *Configuration Options*.

The sample configuration can also be viewed in `file form`.

Important: The sample configuration file is auto-generated from DEEPaaS when this documentation is built. You must ensure your version of DEEPaaS matches the version of this documentation.

```
[DEFAULT]

#
# From deepaas
#

#
# IP address on which the DEEPaaS API will listen.
#
# The DEEPaaS API service listens on this IP address for incoming
# requests.
#  (string value)
#listen_ip = 127.0.0.1

#
# Port on which the DEEPaaS API will listen.
#
# The DEEPaaS API service listens on this port number for incoming
# requests.
#  (port value)
# Minimum value: 0
# Maximum value: 65535
#listen_port = 5000

#
# Run as an OpenWhisk action.
#
# If this option is set to True DEEPaaS will check if the __OW_API_HOST
# environment variable is set. If it is set, it will run an OpenWhisk Docker
```

(continues on next page)

(continued from previous page)

```
# action listener rather than the DEEPaaS API. If it is not set, it will run
# a DEEPaaS in normal mode.
#
# If you specify this option, the value of 'listen-ip' will be used, but the
# port will is hardcoded to 8080 (as OpenWhisk goes to port 8080). Note that
# if you are running inside a container, the most sensible option is to set
# listen-ip to 0.0.0.0
#  (boolean value)
#openwhisk_detect = false

#
# Enable debug endpoint. If set we will provide all the information that you
# print to the standard output and error (i.e. stdout and stderr) through the
# "/debug" endpoint. Default is to not provide this information. This will not
# provide logging information about the API itself.
#  (boolean value)
#debug_endpoint = false

#
# Specify the number of workers to spawn. If using a CPU you probably want to
# increase this number, if using a GPU probably you want to leave it to 1.
# (defaults to 1)
#  (integer value)
#workers = 1

#
# Client's maximum size in a request, in bytes. If a POST request exceeds this
# value, it raises an HTTPRequestEntityTooLarge exception. If set to 0, no
# file size limit will be enforced.
#  (integer value)
# Minimum value: 0
#client_max_size = 0

#
# Pre-warm the modules (eg. load models, do preliminary checks, etc). You might
# want to disable this option if DEEPaaS is loading more than one module
# because
# you risk getting out of memory errors.
#  (boolean value)
#warm = true

#
# From oslo.log
#

# If set to true, the logging level will be set to DEBUG instead of the default
# INFO level. (boolean value)
# Note: This option can be changed without restarting.
#debug = false

# The name of a logging configuration file. This file is appended to any
# existing logging configuration files. For details about logging configuration
```

(continues on next page)

(continued from previous page)

```
# files, see the Python logging module documentation. Note that when logging
# configuration files are used then all logging configuration is set in the
# configuration file and other logging configuration options are ignored (for
# example, log-date-format). (string value)
# Note: This option can be changed without restarting.
# Deprecated group/name - [DEFAULT]/log_config
#log_config_append = <None>

# Defines the format string for %%(asctime)s in log records. Default:
# %(default)s . This option is ignored if log_config_append is set. (string
# value)
#log_date_format = %Y-%m-%d %H:%M:%S

# (Optional) Name of log file to send logging output to. If no default is set,
# logging will go to stderr as defined by use_stderr. This option is ignored if
# log_config_append is set. (string value)
# Deprecated group/name - [DEFAULT]/logfile
#log_file = <None>

# (Optional) The base directory used for relative log_file  paths. This option
# is ignored if log_config_append is set. (string value)
# Deprecated group/name - [DEFAULT]/logdir
#log_dir = <None>

# Uses logging handler designed to watch file system. When log file is moved or
# removed this handler will open a new log file with specified path
# instantaneously. It makes sense only if log_file option is specified and
# Linux platform is used. This option is ignored if log_config_append is set.
# (boolean value)
#watch_log_file = false

# Use syslog for logging. Existing syslog format is DEPRECATED and will be
# changed later to honor RFC5424. This option is ignored if log_config_append
# is set. (boolean value)
#use_syslog = false

# Enable journald for logging. If running in a systemd environment you may wish
# to enable journal support. Doing so will use the journal native protocol
# which includes structured metadata in addition to log messages.This option is
# ignored if log_config_append is set. (boolean value)
#use_journal = false

# Syslog facility to receive log lines. This option is ignored if
# log_config_append is set. (string value)
#syslog_log_facility = LOG_USER

# Use JSON formatting for logging. This option is ignored if log_config_append
# is set. (boolean value)
#use_json = false

# Log output to standard error. This option is ignored if log_config_append is
# set. (boolean value)
```

(continues on next page)

(continued from previous page)

```
#use_stderr = false

# Log output to Windows Event Log. (boolean value)
#use_eventlog = false

# The amount of time before the log files are rotated. This option is ignored
# unless log_rotation_type is set to "interval". (integer value)
#log_rotate_interval = 1

# Rotation interval type. The time of the last file change (or the time when
# the service was started) is used when scheduling the next rotation. (string
# value)
# Possible values:
# Seconds - <No description provided>
# Minutes - <No description provided>
# Hours - <No description provided>
# Days - <No description provided>
# Weekday - <No description provided>
# Midnight - <No description provided>
#log_rotate_interval_type = days

# Maximum number of rotated log files. (integer value)
#max_logfile_count = 30

# Log file maximum size in MB. This option is ignored if "log_rotation_type" is
# not set to "size". (integer value)
#max_logfile_size_mb = 200

# Log rotation type. (string value)
# Possible values:
# interval - Rotate logs at predefined time intervals.
# size - Rotate logs once they reach a predefined size.
# none - Do not rotate log files.
#log_rotation_type = none

# Format string to use for log messages with context. Used by
# oslo_log.formatters.ContextFormatter (string value)
#logging_context_format_string = %(asctime)s.%(msecs)03d %(process)d %(levelname)s
↪%(name)s [%(global_request_id)s %(request_id)s %(user_identity)s] %(instance)s
↪%(message)s

# Format string to use for log messages when context is undefined. Used by
# oslo_log.formatters.ContextFormatter (string value)
#logging_default_format_string = %(asctime)s.%(msecs)03d %(process)d %(levelname)s
↪%(name)s [-] %(instance)s%(message)s

# Additional data to append to log message when logging level for the message
# is DEBUG. Used by oslo_log.formatters.ContextFormatter (string value)
#logging_debug_format_suffix = %(funcName)s %(pathname)s:%(lineno)d

# Prefix each line of exception output with this format. Used by
# oslo_log.formatters.ContextFormatter (string value)
```

(continues on next page)

(continued from previous page)

```
#logging_exception_prefix = %(asctime)s.%(msecs)03d %(process)d ERROR %(name)s
↪%(instance)s

# Defines the format string for %(user_identity)s that is used in
# logging_context_format_string. Used by oslo_log.formatters.ContextFormatter
# (string value)
#logging_user_identity_format = %(user)s %(project)s %(domain)s %(system_scope)s %(user_
↪domain)s %(project_domain)s

# List of package logging levels in logger=LEVEL pairs. This option is ignored
# if log_config_append is set. (list value)
#default_log_levels = amqp=WARN,amqplib=WARN,boto=WARN,qpid=WARN,sqlalchemy=WARN,
↪suds=INFO,oslo.messaging=INFO,oslo_messaging=INFO,iso8601=WARN,requests.packages.
↪urllib3.connectionpool=WARN,urllib3.connectionpool=WARN,websocket=WARN,requests.
↪packages.urllib3.util.retry=WARN,urllib3.util.retry=WARN,keystonemiddleware=WARN,
↪routes.middleware=WARN,stevedore=WARN,taskflow=WARN,keystoneauth=WARN,oslo.cache=INFO,
↪oslo_policy=INFO,dogpile.core.dogpile=INFO

# Enables or disables publication of error events. (boolean value)
#publish_errors = false

# The format for an instance that is passed with the log message. (string
# value)
#instance_format = "[instance: %(uuid)s] "

# The format for an instance UUID that is passed with the log message. (string
# value)
#instance_uuid_format = "[instance: %(uuid)s] "

# Interval, number of seconds, of log rate limiting. (integer value)
#rate_limit_interval = 0

# Maximum number of logged messages per rate_limit_interval. (integer value)
#rate_limit_burst = 0

# Log level name used by rate limiting: CRITICAL, ERROR, INFO, WARNING, DEBUG
# or empty string. Logs with level greater or equal to rate_limit_except_level
# are not filtered. An empty string means that all levels are filtered. (string
# value)
#rate_limit_except_level = CRITICAL

# Enables or disables fatal status of deprecations. (boolean value)
#fatal_deprecations = false
```

5.3 Available commands

PYTHON MODULE INDEX

INDEX