mljet

Release 0.6.0

Konstantin Templin, Kristina Zheltova

Jun 05, 2023

CONTENTS

MLJET

CHAPTER

ONE

MLJET

MLJET - a minimalistic tool for automatic mljetnt of machine learning models.

1.1 Key Features

- Light-weight, cross-platform
- Simple pythonic interface
- Ability to wrap models locally in a service or in a docker container
- Support for multiple model formats
- Support for difference web-frameworks
- Independence of final projects from this tool

1.2 Pipeline

Fig. 1: Pipeline

- First, we initialize the project directory for the next steps;
- Next, we serialize your machine learning models (for example, with Joblib or Pickle);
- Next, we create a final .py file based on the templates that contains the endpoint handlers. Handlers are chosen based on models, and templates based on your preferences (templates are also .py files using, for example, Sanic or Flask);
- Copy or additionally generate the necessary files (e.g. Dockerfile);
- The next step is to compile the API documentation for your project;
- After these steps, we build a Docker container, or a Python package, or we just leave the final directory and then we can deploy your project in Kubernetes, or in Heroku.

1.3 Code example

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

from mljet import cook

X, y = load_iris(return_X_y=True, as_frame=True)

clf = RandomForestClassifier()
clf.fit(X, y)

cook(strategy="docker", model=clf, port=5001)
```

After running script you can see new Docker container. To interact with service simply use CURL:

```
curl -X POST "http://127.0.0.1:5001/predict" -H  "accept: application/json" -H  "Content-
↪Type: application/json" -d '{\"data\":[[5.8, 2.7, 3.9, 1.2]]}'
```

1.4 Communication

- GitHub Issues for bug reports, feature requests and questions.

1.5 License

MIT License (see LICENSE).

1.6 Authors

Templin Konstantin <1qnbhd@gmail.com>

Kristina Zheltova <masterkristall@gmail.com>

1.6.1 Installation

MLJET supports Python 3.8 or newer.

We recommend to install mljet via pip:

```
$ pip install mljet
```

Alternatively, you can install mljet from source:

```
$ git clone git@github.com:qnbhd/mljet.git
$ pip install flit
$ flit build
```

1.6.2 Tutorial

If you are new to MLJET or want a general introduction read this tutorial.

Features

Showcases MLJET features.

Features

Showcases MLJET features.

1. Lightweight and versatile

MLJET is written entirely in Python with few dependencies. This means that once you get interested in MLJET, we can quickly move to a practical example.

Simple Sklearn Local Example

MLJET provides a simple interface to create project and deploy a model.

- *mljet.contrib.entrypoint.cook()* is a function that takes a model and a strategy and deploys it.

In this example, we simply create a project with scikit-learn.

Firstly, import *mljet*.

```
import mljet
```

Import sklearn.ensemble.RandomForestClassifier as classifier and sklearn.datasets to load the iris dataset.

```
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier
```

Let's load dataset and create and train a simple model.

```
X, y = load_iris(return_X_y=True)
clf = RandomForestClassifier()
clf.fit(X, y)
```

Now, we can deploy the model with *mljet.contrib.entrypoint.cook()*. Main arguments are `model` and `strategy`.

The strategy can be either *local* or *docker*. The *local* strategy will deploy the model locally. The *docker* strategy will deploy the model in a docker container. The *mljet.contrib.entrypoint.cook()* function will return a bool or container name.

Now we make only a project without running it. After calling the *mljet.contrib.entrypoint.cook()* function You can see *build* folder in the current directory.

It contains:

- *Dockerfile* - Dockerfile for the model

- *requirements.txt* - requirements for the model
- *models* directory - directory with the dumped model
- *data* directory - directory with the example for the model
- *server.py* - main file for the model

```
mljet.contrib.cook(strategy="local", model=clf)
```

Let's see on *mljet.contrib.entrypoint.cook()* signature. This function accepts a lot of parameters, but we see only the most important ones.

- *model* - model to deploy
- *strategy* - strategy to use
- *backend* - backend to use
- *need_run* - run service after build or not
- *scan_path* - path to scan for requirements
- *silent* - silent mode
- *verbose* - verbose mode

Model parameter is the most important one. It can be any model that implements the *predict* and other methods.

Note: The model must be picklable.

Note: Now is supported *sklearn*, *xgboost*.

Strategy parameter determines the strategy to use.

Backend parameter determines the backend to use. Now is implemented `sanic` and `flask` backends.

Total running time of the script: (0 minutes 0.000 seconds)

2. Deploy with Docker

In this example, we deploy the model with Docker. For this example you need to install Docker, see Docker and XGBoost, see XGBoost.

Firstly, import *mljet*.

```
import mljet
```

Import `xgboost.XGBClassifier` as classifier and `sklearn.datasets` to load the iris dataset.

```
from sklearn.datasets import load_iris
from xgboost import XGBClassifier
```

Let's load dataset and create and train a simple model.

```
X, y = load_iris(return_X_y=True)
clf = XGBClassifier()
clf.fit(X, y)
```

Now, we can deploy the model to *docker* with *mljet.contrib.entrypoint.cook()*.

```
mljet.contrib.cook(
    model=clf,
    strategy="docker",
    tag="mljet-xgboost",
    port=5000,
    need_run=True,
    silent=True,
    verbose=False,
)
```

Let's see on passed parameters.

- *model* - model to deploy - *clf* (XGBoost model)
- *strategy* - strategy to use - *docker*
- *tag* - tag for the docker image - *mljet-xgboost*
- *port* - port for the docker container - *8000*
- *backend* - backend to use - *sanic*, see Sanic
- *need_run* - run service after build or not - *True* (only create container)
- *silent* - silent mode - *True*, non-blocking mode
- *verbose* - verbose mode - *True*, print DEBUG logs

After calling the *mljet.contrib.entrypoint.cook()* function You can see *build* folder in the current directory. And you can see the docker image and container with name *mljet-xgboost*.

Now we can send a request to the model. For this example, we use requests, see Requests. You can use any other tool, for example Postman. Firstly, import requests.

```
import time

import requests
```

Let's sleep for 5 seconds and check the response.

```
time.sleep(5)

response = requests.post(
    "http://localhost:5000/predict",
    json={"data": X.tolist()},
)

print(response.json())
```

Total running time of the script: (0 minutes 0.000 seconds)

1.6.3 Developer's Guide

This document is intended to help developers who want to contribute to the project. It is not intended to be a tutorial on how to use the library, but rather a guide to the codebase.

The mljet project is written in Python and can build and deploy services from different ML models.

The project is structured as follows:

```
├── mljet
│   ├── contrib               -- Create full project from a template, deploy to targets
│   ├── cookie                -- Build main project file
│   └── utils                 -- Utility functions
├── docs                      -- Documentation
│   ├── images                -- Images used in the documentation
│   ├── make.bat              -- Windows build script
│   ├── Makefile              -- Makefile with commands like `make html` to build the␣
↪docs
│   └── source                -- Source files for the documentation
├── examples                  -- Examples of how to use the library
├── LICENSE                   -- License file
├── pyproject.toml            -- Project configuration file
├── README.md                 -- Readme file
├── renovate.json             -- Renovate configuration file
├── requirements-dev.txt      -- Requirements for development
├── requirements-docs.txt     -- Requirements for documentation
├── requirements.txt          -- Requirements for the project
├── setup.cfg                 -- Configuration file for setup.py
├── codecov.yml               -- Configuration file for codecov
└── tests                     -- Tests for the project
```

For run project you need to install requirements.txt and requirements-dev.txt

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

For run tests you need exec:

```
pytest .
```

For run tests with coverage you need exec:

```
pytest --log-level=INFO --cov=mljet --cov-report=xml
```

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:

Report Bugs

Report bugs at 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.

Fix Bugs

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

Implement Features

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

Write Documentation

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

Submit Feedback

The best way to send feedback is to file an issue at 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.

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

Workflow

1. Fork the *mljet* repo on GitHub.

2. Clone your fork locally:

```
$ git clone
```

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

```
$ cd mljet/
$ python3 -m venv env
$ source env/bin/activate
$ pip install -e .
```

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 black, isort, flake8, mypy and the tests, including testing other Python versions with tox:

```
$ black .
$ isort mljet/
$ flake8 .
$ mypy .
$ pytest
```

To get flake8 and mypy to run automatically, consider installing a plugin for your editor. 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.

Commit message guidelines

We use semantic-release to automate the release process. This requires that commit messages are formatted correctly. Please read the semantic-release documentation for more information.

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.md.
3. The pull request should work for Python 3.8, 3.9 and 3.10. Check our CI configuration for the exact versions that are tested.
4. If the pull request adds or changes a command line interface, it should include an example of how to use it in the docstring.
5. Pull requests should be based on the `main` branch.
6. If you are adding a new dependency, please make sure it is added to `requirements.txt` and `requirements-dev.txt`.

Code style guidelines

We use black, isort, flake8, mypy and pylint to enforce a consistent code style. Please make sure your code is compliant by running these tools before submitting a pull request.

```
black .
isort mljet/
flake8 .
mypy .
pylint .
```

Functional programming

We use functional programming to make the code more readable and maintainable. This means that we avoid using mutable variables and side-effects as much as possible. This also means that we prefer functions over classes, and that we prefer immutable data structures like tuples and namedtuples over mutable data structures like lists and dictionaries.

We use the returns library to make working with functional programming in Python easier. Please read the documentation for this library before contributing.

It is worth noting, however, that we try to make sure that the return value and arguments of functions written in functional style are not containerized. This will simplify testing, and also gives ease of transition of the function to imperative style.

Docstrings

We use Google style docstrings. Please make sure your docstrings are compliant by running pydocstyle before submitting a pull request.

```
pydocstyle .
```

Cookie

The *Cookie* module is needed to validate and create final .py files with template-based services.

Cookie has next structure:

```
├── cutter.py        -- Cutter for create final .py files
├── templates        -- Templates for services
│   ├── backends     -- Folder with backends templates
│   └── ml           -- Folder with ML model's wrappers
└── validator.py     -- Validator for check correctness of service
```

Templates

Backends

This directory contains *.py* templates for backends based on different frameworks, such as *flask*, *sanic*, or *fastapi*.

Using exactly regular files allows you to use the full power of static analyzers, formatter and linters. It also makes them easy to run without any extra steps.

This approach also makes it very easy to create "clean" files without any extra effort.

All templates must obey some specification.

Template specification:

- template should have __main__ entrypoint.
- template should have methods to replace, associated with passed methods.
- template should have associated methods-endpoints.
- template should have typing, that is pass mypy check.

Example of template:

```
import ... # import all needed modules

loaded_model = ...

# model wrappers
# need to be replaced with real model wrappers
# can be simple stubs or real wrappers
def predict(model, data) -> list: ...
def predict_proba(model, data) -> list: ...

# framework-specific endpoints
```

(continues on next page)

(continued from previous page)

```
# endpoints functions name must must be marked with a
# special prefix, which is specified in the file
# in method `_get_assoc_endpoint_name`
# now it is `_<model method>`
# for example for `predict` we have `_predict`
@app.post("/predict")
def _predict(...) -> ...: ...

# entrypoint
if __name__ == "__main__":
    ...
```

The logic from the *dispatcher.py* file is used to dynamically search for existing backends by name.

It is also worth noting that each template must have a *Dockerfile* as well as a dependency file.

ML

This directory contains *.py* templates for ML models wrappers.

Let's look at the file for an sklearn model:

```
"""Module that contains Scikit-learn model method's wrappers."""
from mljet.contrib.supported import ModelType

# This constant is needed to determine the type of model
USED_FOR = [ModelType.SKLEARN_MODEL, ModelType.SKLEARN_PIPE]

# this method need to cast numpy array to list
# for `predict` method
def predict(model, data) -> list:
    return model.predict(data).tolist()

# this method need to cast numpy array to list
# for `predict_proba` method
def predict_proba(model, data) -> list:
    return model.predict_proba(data).tolist()
```

At the build stage, these functions replace the functions in the files that describe the backends.

Validator

The validator is used to check templates for compliance with the specification, described above.

Cutter

Templates undergo the following checks before being assembled:

- Validation from *validator* module.
- Static-typing check with *mypy*.

After the checks, the final files are assembled directly based on the template.

- Model's wrappers replacement.
- Import's insertion.
- *Black* formatting.
- *Isort* formatting.

Replacing a function with an appropriate one is done by searching for a regular expression. All the logic for this is in the function *replace_functions_by_names*. It also checks to see if the number of arguments matches, and if the function names match.

Contrib

MLJET *contrib* directory contains a set of scripts needed for final project build and mljetnt.

Now is supported next targets:

- **Local** - build and deploy project to local machine (save or run project)
- **Docker** - build and deploy project to Docker container

Contrib has next structure:

```
├── analyzer.py             -- Module for analyze ML model's methods
├── docker_                 -- Docker target folder
│   ├── docker_builder.py   -- Docker mljetnt target
│   └── runner.py           -- Docker runner
├── entrypoint.py           -- Main project entrypoint
├── local.py                -- Local mljetnt target
├── project_builder.py      -- Project builder
├── supported.py            -- List of supported models, targets, etc.
└── validator.py            -- Project validator
```

Description

- **analyzer.py** - Module for analyze ML model's methods (extract methods names, etc.) and find associated wrapper to paste it into backend template.
- **docker/_** - folder with Docker target scripts
- **docker/_builder.py** - script for build and deploy project to Docker container
- **runner.py** - Module for build, run and stop Docker container
- **entrypoint.py** - Main project entrypoint, contains main *cook* function
- **local.py** - Module for build and deploy project to local machine
- **project_builder.py** - Module for build project

- **supported.py** - List of supported models, targets, etc.
- **validator.py** - Module for validate project

Testing

The test suite is run with the following command:

```
$ pytest .
```

Tests directory has the following structure:

```
├── conftest.py         -- Pytest configuration file
├── contrib             -- Tests for contrib modules
├── cookie              -- Tests for `cookie` module
├── functional          -- Functional tests
├── iomock.py           -- Mocks for IO
├── test_version.py     -- Version test
└── utils               -- Tests for `utils` module
```

As metric for testing, we use code coverage. We also occasionally run mutation testing to check for possible bugs in the test suite. For mutation testing we use cosmic-ray.

As an additional library for testing we use hypothesis. It allows us to write tests that generate random data and check that our code works correctly with it. We believe that this approach is more effective than writing tests for specific cases.

IO Mock

This functionality requires a separate description. It is used to mock the input and output streams. It is used in the tests to check the correctness of the output of the program.

The main idea is to replace the standard input and output streams with the streams that are controlled by the test. The test can write data to the input stream and read data from the output stream.

You can see file iomock.py.

IOMock class mock the filesystem for testing purposes.

The filesystem is represented as a tree, where each node is a dict. The keys are the names of the files and directories, and the values are either dicts (for directories) or strings (for files).

The tree is stored in the *tree* attribute, and the root of the tree is stored in the *start_point* attribute.

The *to_forward* argument is a list of paths to files that will be copied to the mock filesystem. This is useful for testing code that reads from the filesystem.

The *fs_tree* property returns a string representation of the tree.

After tests we can check that the tree is the same as the original one.

```
def test_default_iomock():
"""Test the default IO mock."""
    # Create a mock object
    mocker = DefaultIOMock()
    # Get into io-mock context
    with mocker:
        # write to file
```

(continues on next page)

(continued from previous page)

```
        with open("test.txt", "w") as fo:
            fo.write("Hello, world!")
        # read from file
        with open("test.txt", "r") as fi:
            assert fi.read() == "Hello, world!"
        # write to binary file
        with open("model.pkl", "wb") as fo:
            fo.write(b"Hello, world!")
        # read from binary file
        with open("model.pkl", "rb") as fi:
            assert fi.read() == b"Hello, world!"
        # Create a directory
        Path("a/b/c").mkdir(parents=True, exist_ok=True)
        assert Path("test.txt").exists()
        assert Path("model.pkl").exists()
    assert (
        mocker.fs_tree
        == """|
    └── tests
    |    ├── a
    |        ├── b
    |            ├── c
    |    ├── model.pkl
    |    ├── test.txt
    """
    )
```

1.6.4 API Reference

mljet

Subpackages

mljet.contrib

Subpackages

mljet.contrib.docker_

Submodules

mljet.contrib.docker_.docker_builder module

mljet.contrib.docker_.runner module

Module contents

Submodules

mljet.contrib.analyzer module

Models analyzed and method's extractor.

mljet.contrib.analyzer.**extract_methods_names**(*model*)

Get methods from model.

Return type
List[str]

mljet.contrib.analyzer.**get_associated_methods_wrappers**(*model*)

Get methods names and associated wrappers.

Return type
Dict[str, Callable]

mljet.contrib.entrypoint module

Docker builder module.

mljet.contrib.entrypoint.**cook**(**, model, strategy=Strategy.DOCKER, backend=None, tag=None, base_image=None, container_name=None, need_run=True, port=None, scan_path=None, n_workers=1, silent=True, verbose=False, remove_project_dir=False, ignore_mypy=False, additional_requirements_files=None*)

Cook web-service.

Parameters

- **model** – model to deploy
- **strategy** (Union[*Strategy*, str]) – strategy to use
- **backend** (Union[str, Path, None]) – backend to use
- **tag** (Optional[str]) – tag for docker image
- **base_image** (Optional[str]) – base image for docker image
- **container_name** (Optional[str]) – container name
- **need_run** (bool) – run service after build or not
- **port** (Optional[int]) – port to use
- **scan_path** (Union[str, Path, None]) – path to scan for requirements
- **n_workers** (int) – number of workers
- **silent** (bool) – silent mode
- **verbose** (bool) – verbose mode
- **remove_project_dir** (bool) – remove project directory after build
- **ignore_mypy** (bool) – ignore mypy errors
- **additional_requirements_files** (Optional[Sequence[Union[str, Path]]]) – additional requirements files

Return type
Dict[str, Any]

Returns
Result of build, maybe bool or container name (if docker strategy)

mljet.contrib.local module

mljet.contrib.project_builder module

Project builder.

mljet.contrib.project_builder.**build_backend**(*path*, *filename*, *template_path*, *models*, *imports=None*, *ignore_mypy=False*)

Return type
Path

mljet.contrib.project_builder.**build_requirements_txt**(*project_path*, *backend_path*, *scan_path*, *additional_requirements_files=None*)

Builds requirements.txt

Return type
Path

mljet.contrib.project_builder.**copy_backend_dockerfile**(*project_path*, *backend_path*)

Copies backend Dockerfile to project_path.

Return type
Path

mljet.contrib.project_builder.**dumps_models**(*path*, *models*, *models_names*, *serializer=<module 'pickle' from '/home/docs/.pyenv/versions/3.8.6/lib/python3.8/pickle.py'>*, *ext='pkl'*)

Dumps models to models_path.

Return type
Path

mljet.contrib.project_builder.**full_build**(*project_path*, *backend_path*, *template_path*, *scan_path*, *models*, *models_names*, *filename='server.py'*, *imports=None*, *serializer=<module 'pickle' from '/home/docs/.pyenv/versions/3.8.6/lib/python3.8/pickle.py'>*, *ext='pkl'*, *ignore_mypy=False*, *additional_requirements_files=None*)

Builds project.

Return type
Result[Path, Exception]

mljet.contrib.project_builder.**init_project_directory**(*path*, *force=False*)

Initializes project directory.

Return type
Path

mljet.contrib.project_builder.**managed_write**(*filepath*, *writer*, *mode='w'*)

Writes obj to stream using writer.

Return type
Result[TypeVar(_ValueType, covariant=True), Exception]

mljet.contrib.supported module

Supported models types, strategies.

class mljet.contrib.supported.**ModelType**(*value*)

Bases: str, Enum

Model type.

CATBOOST = 'catboost'

LAMA = 'lightautoml'

LGBM = 'lightgbm'

SKLEARN = 'sklearn'

XGBOOST = 'xgboost'

classmethod from_model(*model*)

Get model type from model.

Parameters
model (*Estimator*) – model to get type from

Returns
Model type.

class mljet.contrib.supported.**Strategy**(*value*)

Bases: str, Enum

An enumeration.

DOCKER = 'DOCKER'

LOCAL = 'LOCAL'

mljet.contrib.validator module

Module, that contains validators for mljet.

mljet.contrib.validator.**validate_ret_backend**(*backend*)

Validates predefined backend name or path to custom backend.

Return type
Path

mljet.contrib.validator.**validate_ret_container_name**(*name*)

Validates container name and returns it if it is valid.

Return type
str

mljet.contrib.validator.**validate_ret_model**(*model*)

Validates model and returns it type if it is valid.

Return type
ModelType

mljet.contrib.validator.**validate_ret_port**(*port*)

Validates port and returns it if it is valid.

Return type
int

mljet.contrib.validator.**validate_ret_strategy**(*strategy*)

Validates strategy and returns it if it is valid.

Return type
Strategy

Module contents

Module for docker related functions.

mljet.cookie

Subpackages

mljet.cookie.templates

Subpackages

mljet.cookie.templates.backends

Submodules

mljet.cookie.templates.backends.dispatcher module

Dispatch for default backends.

mljet.cookie.templates.backends.dispatcher.**dispatch_default_backend**(*backend_name*, *strict=False*)

Dispatches default backend by name.

Return type
Optional[Path]

mljet.cookie.templates.backends.dispatcher.**get_all_default_backends**()

Returns all default backends.

Return type
Dict[str, Path]

Module contents

mljet.cookie.templates.ml

Submodules

mljet.cookie.templates.ml.dispatcher module

Dispatcher for supported model types.

mljet.cookie.templates.ml.dispatcher.**get_all_supported_ml_kinds**()

Returns all default backends.

Return type
Dict[str, ModuleType]

mljet.cookie.templates.ml.dispatcher.**get_dual_methods**(*mt*, *methods*)

Get dual methods, needed to replace in backend templates.

Return type
List[Callable]

Module contents

Module contents

Submodules

mljet.cookie.cutter module

Module that contains app builder.

exception mljet.cookie.cutter.**MypyValidationError**

Bases: Exception

Exception raised when the template is not passing mypy check.

args

mljet.cookie.cutter.**build_backend**(*template_path*, *methods_to_replace*, *methods*, *imports=None*, *ignore_mypy=False*)

Build app from template.

Parameters

- **template_path** (Union[str, Path]) – path to template.
- **methods_to_replace** (Sequence[str]) – methods to replace in template.
- **methods** (Sequence[Callable]) – methods to replace with.
- **imports** (Optional[Sequence[str]]) – imports to insert into template.
- **ignore_mypy** (bool) – ignore mypy check.

Return type
str

Returns
Result with app source code.

Template specification:

- template should have __main__ entrypoint.
- template should have methods to replace, associated with passed methods.
- template should have associated methods-endpoints.
- template should have typing, that is pass mypy check.

Some:
Reporting intermediate ddd data such as the current trial number back to the framework, as done in `MyPyValidationError`.

Raises

- ***MypyValidationError*** – if template is not passing mypy check.
- ***ValidationError*** – if template is not passing validation.
- **TypeError** – if template is not passing validation.
- **FileNotFoundError** – if template is not found.

Note: After app is built, it should be formatted with black, isort.

`mljet.cookie.cutter.`**insert_import**(*text*, *deps*)

Inserts import into text.

Parameters

- **text** (str) – text to insert import into.
- **deps** (Sequence[str]) – imports to insert.

Return type
str

Returns
Text with inserted import.

`mljet.cookie.cutter.`**mypy_run**(*text*)

Run mypy check on template.

Parameters
text (str) – Source code of template.

Return type
str

Returns
Result with mypy output.

`mljet.cookie.cutter.`**replace_functions_by_names**(*source*, *names2repls*)

Replace functions by names in source code with passed functions.

Return type
str

mljet.cookie.validator module

Static code analysis of the template.

exception mljet.cookie.validator.**ValidationError**

Bases: Exception

Exception raised when the template is not valid.

args

mljet.cookie.validator.**validate**(*source*, *methods*)

Validate the template.

Parameters

- **source** (str) – The source code of the template
- **methods** (Sequence[str]) – Sequence of the methods

Return type
bool

Returns
True if the template is valid, False otherwise

Module contents

mljet.utils

Submodules

mljet.utils.conn module

Helper functions for connecting across the network.

mljet.utils.conn.**find_free_port**()

Find a free port.

Return type
int

Returns
Number of free port to use.

mljet.utils.conn.**is_port_in_use**(*port*)

Check if port is in use.

Parameters
port (int) – port to check

Return type
bool

Returns
True if port is in use, False otherwise

mljet.utils.logging_ module

Logging module.

class mljet.utils.logging_.**RichEmojiFilteredHandler**(**args*, *enable_emoji=False*, ***kwargs*)

Bases: RichHandler

Extended rich handler with emoji filter support.

Initializes the instance - basically setting the formatter to None and the filter list to empty.

HIGHLIGHTER_CLASS

alias of ReprHighlighter

KEYWORDS = ['GET', 'POST', 'HEAD', 'PUT', 'DELETE', 'OPTIONS', 'TRACE', 'PATCH']

acquire()

Acquire the I/O thread lock.

addFilter(*filter*)

Add the specified filter to this handler.

close()

Tidy up any resources used by the handler.

This version removes the handler from an internal map of handlers, _handlers, which is used for handler lookup by name. Subclasses should ensure that this gets called from overridden close() methods.

createLock()

Acquire a thread lock for serializing access to the underlying I/O.

emit(*record*)

Invoked by logging.

Return type
None

filter(*record*)

Determine if a record is loggable by consulting all the filters.

The default is to allow the record to be logged; any filter can veto this and the record is then dropped. Returns a zero value if a record is to be dropped, else non-zero.

Changed in version 3.2: Allow filters to be just callables.

flush()

Ensure all logging output has been flushed.

This version does nothing and is intended to be implemented by subclasses.

format(*record*)

Extends RichHandler. Format to filter out emoji.

Return type
str

get_level_text(*record*)

Get the level name from the record.

Parameters
record (*LogRecord*) – LogRecord instance.

Returns
A tuple of the style and level name.

Return type
Text

`get_name()`

`handle(record)`

Conditionally emit the specified logging record.

Emission depends on filters which may have been added to the handler. Wrap the actual emission of the record with acquisition/release of the I/O thread lock. Returns whether the filter passed the record for emission.

`handleError(record)`

Handle errors which occur during an emit() call.

This method should be called from handlers when an exception is encountered during an emit() call. If raiseExceptions is false, exceptions get silently ignored. This is what is mostly wanted for a logging system - most users will not care about errors in the logging system, they are more interested in application errors. You could, however, replace this with a custom handler if you wish. The record which was being processed is passed in to this method.

`property name`

`release()`

Release the I/O thread lock.

`removeFilter(filter)`

Remove the specified filter from this handler.

`render(*, record, traceback, message_renderable)`

Render log for display.

Parameters

- **record** (*LogRecord*) – logging Record.
- **traceback** (*Optional[Traceback]*) – Traceback instance or None for no Traceback.
- **message_renderable** (*ConsoleRenderable*) – Renderable (typically Text) containing log message contents.

Returns
Renderable to display log.

Return type
ConsoleRenderable

`render_message(record, message)`

Render message text in to Text.

Parameters

- **record** (*LogRecord*) – logging Record.
- **message** (*str*) – String containing log message.

Returns
Renderable to display log message.

Return type
ConsoleRenderable

`setFormatter`(*fmt*)

Set the formatter for this handler.

`setLevel`(*level*)

Set the logging level of this handler. level must be an int or a str.

`set_name`(*name*)

`mljet.utils.logging_.init`(*verbose=False*, *enable_emoji=False*, *rich=True*)

Init logging.

Args:

verbose (bool):
Verbose level (DEBUG) or not (INFO).

enable_emoji (bool):
Enable emoji in logs or not.

rich (bool):
Enable rich handler & traceback or not.

Raises:
AnyError: If anything bad happens.

mljet.utils.names_generator module

Docker image name generator.

`mljet.utils.names_generator.get_random_name`()

Return type
str

mljet.utils.requirements module

Module for scanning Python files for requirements.

`mljet.utils.requirements.extract_modules`(*node*, *ignore_mods=None*)

Extract the modules from an import node

Parameters

- **node** (Union[Import, ImportFrom]) – The import node
- **ignore_mods** (Optional[List[str]]) – List of modules to ignore

Return type
dict

`mljet.utils.requirements.freeze`()

Get a dictionary of installed packages and their versions

Return type
Dict[str, str]

Returns
A dictionary of installed packages and their versions

mljet.utils.requirements.**get_pkgs_distributions**()

Get a dictionary of installed packages and their module names

Return type
dict

Returns
A dictionary of installed packages and their module names

mljet.utils.requirements.**get_source_from_notebook**(*path*)

Extract the source code from a Jupyter notebook

Parameters
path (Union[str, Path]) – Path to the notebook

Return type
str

Returns
The source code as a string

Raises
RuntimeError – If the notebook is not valid JSON

mljet.utils.requirements.**make_requirements_txt**(*path*, *out_path='requirements.txt'*, *strict=True*, *extensions=None*, *ignore_mods=None*)

Make a requirements.txt file from a directory of files.

Parameters

- **path** (Union[str, Path]) – Path to the directory
- **out_path** (Union[str, Path]) – Path to the output file
- **extensions** (Optional[List[str]]) – List of file extensions to scan. Defaults to ['py', 'ipynb']
- **strict** (Optional[bool]) – Set only the exact version of the packages
- **ignore_mods** (Optional[List[str]]) – List of modules to ignore

Return type
Dict[str, str]

Returns
A dict of requirements and their versions

Raises
ValueError – If the path is not correct

mljet.utils.requirements.**merge**(**requirements_lists*)

Merge requirements lists.

Parameters
requirements_lists (List[str]) – list of requirements

Return type
List[str]

Returns
Merged requirements

mljet.utils.requirements.**merge_requirements_txt**(**files*, *ignore_prefixes=None*)

Merge requirements.txt files.

Parameters

- **files** (Union[str, Path]) – list of requirements.txt files
- **ignore_prefixes** (Optional[List[str]]) – list of prefixes to ignore

Return type
List[str]

Returns
Final requirements.txt file content

mljet.utils.requirements.**scan_requirements**(*path*, *extensions=None*, *ignore_mods=None*, *ignore_names=None*)

Scan a directory of file for requirements.

Parameters

- **path** (Union[str, Path]) – Path to the directory
- **extensions** (Optional[List[str]]) – List of file extensions to scan. Defaults to ['py', 'ipynb']
- **ignore_mods** (Optional[List[str]]) – List of modules to ignore
- **ignore_names** (Optional[List[str]]) – List of file/dirs names to ignore

Return type
Dict[str, str]

Returns
A dict of requirements and their versions

Raises
ValueError – If the path is not correct

mljet.utils.requirements.**validate**(*req*)

mljet.utils.types module

Module contains type's aliases.

final class mljet.utils.types.**Estimator**(**args*, ***kwargs*)

Bases: Protocol

Protocol for estimators.

fit(**args*, ***kwargs*)

Return type
Any

predict(**args*, ***kwargs*)

Return type
Any

final class mljet.utils.types.**Serializer**(**args*, ***kwargs*)

Bases: Protocol

Protocol for serializers.

static dump(*obj*, *file*, **args*, ***kwargs*)

Return type
None

static load(*file*, **args*, ***kwargs*)

Return type
Any

mljet.utils.utils module

Utils module.

mljet.utils.utils.**drop_unnecessary_kwargs**(*func*, *kwargs*)

Drop unnecessary kwargs.

Return type
dict

mljet.utils.utils.**is_package_installed**(*name*)

Check if package is installed and valid.

Return type
bool

mljet.utils.utils.**parse_cls_name**(*obj*)

Parse class name.

Return type
str

Module contents

Helper functions for the mljet package.

Module contents

MLJET - A simple Open Source mljetnt tool for ML models

If you have been working on ML models, then you have probably faced the task of deploying these models. Perhaps you are participating in a hackathon or want to show your work to management. According to our survey, more than 60% of the data-scientists surveyed faced this task and more than 60% of the respondents spent more than half an hour creating such a service.

The most common solution is to wrap it in some kind of web framework (like Flask).

1.6.5 Changelog

This is a record of all past MLJET releases and what went into them, in reverse chronological order. All previous releases should still be available on PyPI.

- **Merge pull request #146 from qnbhd/lama_support** by *Konstantin T* at *2023-06-05 15:40:58*

  Lama support

- **fix: update lama usage example** by *Templin Konstantin* at *2023-06-05 14:04:25*
- **fix: change flack base_image and lama deploy strategy** by *pacificus* at *2023-04-03 16:08:36*
- **fix: change code style** by *pacificus* at *2023-04-03 15:54:31*
- **feat: add lightautoml to test and build** by *pacificus* at *2023-04-03 15:43:51*
- **feat: add lightautoml prediction wrapper** by *pacificus* at *2023-04-03 15:32:07*
- **feat: add lightautoml support** by *pacificus* at *2023-03-27 22:48:12*
- **Update README.md** by *Konstantin T* at *2023-05-06 16:18:18*
- **Merge branch 'update-docs-v0.0.7'** by *Templin Konstantin* at *2023-05-06 16:04:49*
- **feat: update push release** by *Templin Konstantin* at *2023-05-06 15:15:12*

PYTHON MODULE INDEX

m

INDEX

H

I

K

L

M

N

P

R

S

V

X