yacmmal

Release 0.1.5

Juan S. Lara

Mar 30, 2022

CONTENTS:

Yet Another Config Manager for MAchine Learning (yacmmal) is a package to automatically load config files for machine learning projects. `yacmmal` is built on top of `pydantic` and allows type checking, and automatic creation of dataclasses from different file formats like YAML, JSON, TOML, among others.

CHAPTER

ONE

INSTALLATION

- You can install this package from the PyPi repository:

```
pip install yacmmal
```

Or with `poetry`:

```
poetry add yacmmal
```

- You can also build this package from source (requires `poetry`):

```
git clone https://github.com/juselara1/yacmmal
cd yacmmal
poetry install
```

CHAPTER

TWO

USER GUIDE

2.1 Understanding the Config Dataclass

`yacmmal` uses a `Config` dataclass to load different types of machine learning configurations, this object is created as a composition of the following dataclasses:

- `paths`: it's intended to store different paths that are used in the project, e.g., source data paths, experiments folders, directories to save the models.
- `database`: you can place here databases' related configurations like host name, port, table names.
- `hyperparameters`: this attribute groups the hyperparameters that are typically used in the models, for instance activation functions, number of units per layer, regularization constant, kernel functions, among others.
- `experiment`: this stores experiment-related configurations, like train-test proportions, number of folds for k-fold cross-validation, number of trials, class weights, among others.
- `training`: the training configuration determines the parameter's estimation process, this includes number of epochs, batch size, verbosity, among others.
- `evaluation`: this can contain testing configurations such as thresholds, evaluation metrics, significance levels, among others.
- `optimization`: in some cases (like `tensorflow` or `pytorch`), the model's optimization can be configured, this attribute can be used to store parameters like the optimizer kind, learning rate, burndown rate, among others.
- `deploy`: it's related to the deployment used in the ML application, e.g., routes and ports in an API, computational resources (maximum RAM and jobs), among others.

2.2 The Autoconfig Decorator

The decorator mode allows to extract configurations in a single function. For instance, suppose that you have a project in which you must define a Neural Network and evaluate it, something similar to the following structure:

```
.
├── config
│   ├── ep_file.yaml
│   └── hp_file.yaml
└── main.py
```

- `hp_file.yaml` contains the hyperparameters for your model:

```
activation: "relu"
hidden_units:
  - 32
  - 64
dropout: 0.2
```

- `ep_file.yaml` contains the experiment parameters:

```
test_size: 0.3
k_fold: 5
```

You can easily load these parameters into a single `Config` object using a decorated function:

```
# main.py
from yacmmal import autoconfig, BaseModel
from typing import List

class HyperParams(BaseModel):
    activation: str
    hidden_units: List[int]
    dropout: float

class ExperimentParams(BaseModel):
    test_size: float
    k_fold: int

@autoconfig(
        base_path="config/",
        config=[("hp_file", "hyperparameters", HyperParams),
                ("ep_file", "experiment", ExperimentParams)],
        format="yaml"
        )
def load_cfg(cfg):
    print(cfg)
    ...
```

The `autoconfig` decorator defines:

- `base_path`: for the root path of the config files.
- `config`: a sequence of tuples, such that each tuple contains three elements `(file_name, config_type, dataclass)`. The `config_type` must be one of the `yacmmal.types.config.ConfigAttrs`
- `format`: the file format of the config files, must be one of the `yacmmal.types.formats.ConfigFormat`.

  You can find this example here.

2.3 Building a Config Object

The builder API allows to dynamically build a `Config` object, using the `yacmmal`'s loaders. For instance, suppose that you have a project in which you fetch information from `MySQL` and train a `SVM` model, you could have the following configuration files:

```
.
└── config
    ├── db.json
    └── hp_file.json
```

- The `db.json` file contains the configurations for the database:

```
{
    "hostname": "localhost",
    "port": 3306,
    "user": "root",
    "password": "root",
    "database": "test"
}
```

- The `hp_file.json` contains the model's hyperparameters:

```
{
    "kernel": "rbf",
    "gamma": 0.1,
    "C": 1.0
}
```

You can load this configurations using the builder mode, as follows:

```
# main.py
from yacmmal import BaseModel
from yacmmal.load.json import JSONLoader
from typing import List

class DBParams(BaseModel):
    hostname: str
    port: int
    user: str
    password: str
    database: str

class HyperParams(BaseModel):
    kernel: str
    gamma: float
    C: float

loader = JSONLoader(base_path="config/")
cfg = (
        loader
        .add_path("hp_file", "hyperparameters", HyperParams)
        .add_path("db", "database", DBParams)
```

(continues on next page)

(continued from previous page)

```
        .extract()
        )
print(f"Config: {cfg}")
```

- The `yacmmal.load.base.Loader` is initialized with the `base_path` of the configuration files.
- The `add_path` method receives:
  - `path`: file name for the configuration.
  - `name`: a configuration attribute defined at `yacmmal.types.config.ConfigAttrs`
  - `dclass`: dataclass used to extract the configurations.
- The `extract` method generates a `Config` object with the consolidated configurations.

  You can find this example here.

2.4 Nested Configurations

There are some cases in which you are working in a complex application that contains several configurations for the same attribute. For example, suppose that you need to collect data from multiple databases and you want to define several models. You can address this using dataclasses in composition as We'll show. First, suppose that you have the following configurations:

```
.
└── config
    ├── database.toml
    └── hparams.toml
```

- `database.toml` contains the configurations for two databases:

```toml
[mysql]
host = "localhost"
port = 3306
user = "root"
database = "test"

[postgresql]
host = "localhost2"
port = 5432
user = "postgres"
database = "test"
```

- `hparams.toml` contains the configurations for two models:

```toml
[neural_network]
activation = "relu"
hidden_units = [64, 32]
dropout = 0.5

[svm]
kernel = "rbf"
```

(continues on next page)

(continued from previous page)

```
gamma = 0.1
C = 1.0
```

If you use composition in the dataclasses, you can load the configurations with `yacmmal`:

```
# main.py
from yacmmal import autoconfig, BaseModel
from yacmmal.types.config import Config
from typing import List

class NNParams(BaseModel):
    activation: str
    hidden_units: List[int]
    dropout: float

class SVMParams(BaseModel):
    kernel: str
    gamma: float
    C: float

class HyperParams(BaseModel):
    neural_network: NNParams
    svm: SVMParams

class MySQL(BaseModel):
    host: str
    port: int
    user: str
    database: str

class PostgreSQL(BaseModel):
    host: str
    port: int
    user: str
    database: str

class DBParams(BaseModel):
    mysql: MySQL
    postgresql: PostgreSQL

class Config(BaseModel):
    hyperparams: HyperParams
    database: DBParams

@autoconfig(
        base_path="config/",
        config=[
            ("hparams", "hyperparameters", HyperParams),
            ("database", "database", DBParams)
            ],
        format="toml"
        )
```

(continues on next page)

(continued from previous page)

```
def main(cfg: Config):
    print(cfg)
    print(cfg.database.mysql)
    ...
```

In this example, We defined a custom `Config` object to leverage from Python's duck typing, this allows code linting, type checking and autocompletion in the `main` function.

> You can find this example here.

CHAPTER

THREE

YACMMAL

3.1 yacmmal package

3.1.1 Subpackages

yacmmal.load package

Submodules

yacmmal.load.base module

yacmmal.load.conf module

yacmmal.load.json module

yacmmal.load.toml module

yacmmal.load.yaml module

Module contents

yacmmal.types package

Submodules

yacmmal.types.config module

yacmmal.types.formats module

Module contents

3.1.2 Submodules

3.1.3 yacmmal.base module

3.1.4 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search