PyPads

Release 0.1.20

Jun 22, 2020

Install PyPads:

Building on the MLFlow toolset, PyPaDS aims to extend the existing tracking functionality, make logging as easy as possible for the user. The production of structured results is an additional goal of the extension.

CHAPTER 1

Install PyPads

Logging your experiments manually can be overwhelming and exhaustive? PyPads is a tool to help automate logging as much information as possible by tracking the libraries of your choice.

- **Installing PyPads:** *With pip* | *From source*

1.1 How To Install

There are different ways to install pypads:

- *Install the latest official release*. This is the best approach for most users. It will provide a stable version and pre-built packages are available for most platforms.
- *Building the package from source*. This is best for users who want the latest features and aren't afraid of running brand-new code. This is also needed for users who wish to contribute to the project.

1.1.1 Installing the latest release with pip

The lastest stable version of pypads can be downloaded and installed from PyPi:

```
pip install pypads
```

Note that in order to avoid potential conflicts with other packages it is strongly recommended to use a virtual environment, e.g. python3 `virtualenv` (see python3 virtualenv documentation) or conda environments.

Using an isolated environment makes possible to install a specific version of pypads and its dependencies independently of any previously installed Python packages. In particular under Linux is it discouraged to install pip packages alongside the packages managed by the package manager of the distribution (apt, dnf, pacman. . .).

Note that you should always remember to activate the environment of your choice prior to running any Python command whenever you start a new terminal session.

Warning: Pypads requires Python 3.6 or newer.

1.1.2 Installing pypads from source

This section introduces how to install the **master branch** of pypads. This can be done by building from source.

Building from source

Building from source is required to work on a contribution (bug fix, new feature, code or documentation improvement).

1. Use Git to check out the latest source from the pypads repository on Github.:

   ```
   git clone git@github.com:padre-lab-eu/pypads.git  # add --depth 1 if your␣
   ↪connection is slow
   cd pypads
   ```

   If you plan on submitting a pull-request, you should clone from your fork instead.

2. Install poetry tool for dependency managenment for your platform. See instructions in the Official documentation.:

   ```
   pip install poetry
   ```

3. Optional (but recommended): create and activate a dedicated virtualenv or conda environment.

4. Build the project with poetry, this will generate a whl and a tar file under dist/:

   ```
   poetry build
   ```

5. Install pypads using one of the two generated files:

   ```
   pip install dist/pypads-X.X.X.tar.gz
   OR
   pip install dist/pypads-X.X.X-py3-none-any.whl
   ```

If the package is available on pypi but can't be found with poetry you might want to delete your local poetry cache :

> poetry cache clear –all pypi

Dependencies

Runtime dependencies

Pypads requires the following dependencies both at build time and at runtime:

- Python (>= 3.6),
- cloudpickle (>= 1.3.3),
- mlflow (>= 1.6.0),
- boltons (>= 19.3.0),
- loguru (>=0.4.1)

Those dependencies are **automatically installed by poetry** if they were missing when building pypads from source.

Build dependencies

Building PyPads also requires:

- Poetry >= 0.12.

Test dependencies

Running tests requires:

- pytest >= 5.2.5,
- scikit-learn >= 0.21.3,
- tensorflow >= 2.0.0b1,
- psutil >= 5.7.0,
- networkx >= 2.4,
- keras >= 2.3.1.

Some tests also require numpy.

CHAPTER 2

Getting started

Learn more about how to use pypads, configuring your tracking events and hooks, mapping your custom logging function and some of the core features of PyPads.

- **Usage example** *Decision Tree Iris classification*
- **Mapping file example for Scikit-learn** A *mapping file* is where we define the classes and functions to be tracked from the library of our choice. It includes the defined hooks.
- **Hooks and events**
    - *Events* are defined primarily by listeners which are, in our case, **hooks**. When triggered, the corresponding loggers are called. Logging functions are linked to these events via a mapping dictionary passed to the *base class*.
    - *Hooks* help the user to define what triggers those events (e.g. what functions or classes should trigger a specific event).
- **Loggers** Logging functions are functions called around when any tracked method or class triggers their corresponding event. Mapping events to logging functions is done by passing a dictionary **mapping** as a parameter to the *PyPads class*.

The following tables show the default loggers of pypads.

- **Event Based loggers**

Logger	Event	Hook	Description
LogInit	init	'pypads_init'	Debugging purposes
Log	log	'pypads_log'	Debugging purposes
Parameters	parameters	'pypads_fit'	tracks parameters of the tracked function call
Cpu,Ram,Disk	hardware	'pypads_fit'	track usage information, properties and other info on CPU, Memory and Disk.
Input	input	'pypads_fit'	tracks the input parameters of the current tracked function call.
Output	output	'pypads_predict', 'pypads_fit'	Logs the output of the current tracked function call.
Metric	metric	'pypads_metric'	tracks the output of the tracked metric function.
PipelineTracker	pipeline	'pypads_fit','pypads_predict', 'pypads_transform', 'pypads_metrics'	tracks the workflow of execution of the different pipeline elements of the experiment.

- **Pre/Post run loggers**

Logger	Pre/Post	Description
IGit	Pre	Source code management and tracking
ISystem	Pre	System information (os,version,machine…)
ICpu	Pre	Cpu information (Nbr of cores, max/min frequency)
IRam	Pre	Memory information (Total RAM, SWAP)
IDisk	Pre	Disk information (disk total space)
IPid	Pre	Process information (ID, command, cpu usage, memory usage)
ISocketInfo	Pre	Network information (hostname, ip address)
IMacAddress	Pre	Mac address

2.1 Getting started with PyPads

PyPads is a tracking framework for your python programs. It implements an infrastructure featuring the possibilities for:

- Community driven mapping files
- Logging injection by importlib extension
- Timekeeping
- Full access to the current state in logging functions
- Prefabricated tracking functions and formats
- Data and control flow manipulation with actuators
- Run based data caching for loggers

The framework was developed for machine learning experiments and is based on mlflow. The main focus for PyPads is based in its ulterior, but pythonic manner of use. PyPads aims to deliver a way to harmonize results of a multitude of libraries in a structured way, while stepping out of the way if needed. Most dependencies of PyPads are to be considered as optional and are only used to extend on more sophisticated logging functions.

In its core app, PyPads allows for registering plugin extensions. These can be used to define packages introducing new loggers, validators, actuators, decorators etc.

2.1.1 Quick start

Install PyPads assuming Python 3 is already installed:

```
$ pip install pypads
```

Usage

Activating PyPads for tracking in its default setting is as easy as adding two lines to your experiment.

A simple example looks like the following.

```
from pypads.app.base import PyPads
PyPads(autostart=True)

# An example
from sklearn import datasets, metrics
from sklearn.tree import DecisionTreeClassifier

# load the iris datasets
dataset = datasets.load_iris()

# fit a model to the data
model = DecisionTreeClassifier()
model.fit(dataset.data, dataset.target) # pypads will track the parameters, output, and input of the model fit function.
# get the predictions
predicted = model.predict(dataset.data) # pypads will track only the output of the model predict function.
```

Results

By default results can be found in the `.mlruns` folder in the home directory of the executing user. While this can be changed when initializing the app, you can also specify the environment variable `MLFLOW_PATH` to define a custom location.

2.1.2 Concepts

PyPads includes a set of concepts, of which some are to be followed because of technical reasons, while others only impose semantical meaning.

Actuators

Actuators are features of PyPads manipulating experiments. When using an actuator the result of the experiment may be or is impacted. Actuators can include changes to the underlying machine learning code, setup and more. An exemplary actuator is an actuator enforcing a random seed setup. Custom, new or other actuators can be added to an IActuators plugin exposing them to PyPads.

```
@actuator
def set_random_seed(self, seed=None):
    # Set seed if needed
    if seed is None:
        import random
        # Numpy only allows for a max value of 2**32 - 1
        seed = random.randrange(2 ** 32 - 1)
    self.pypads.cache.run_add('seed', seed)

    from pypads.injections.analysis.randomness import set_random_seed
    set_random_seed(seed)
```

To call an actuator you can use the app.

```
from pypads.app.base import PyPads
tracker = PyPads(autostart=True)
tracker.actuators.set_random_see(seed=1)
```

API

The PyPads API delivers standard functionality of PyPads. This also pipes some of mlflow features. You can start, stop runs, log artifacts, metrics or parameters, set tags and write meta information about them. Additionally the PyPads API inroduces setup and teardown (also called pre and post run) functions to be called and also to manually mark functions for tracking. A full documentation can be found here. To call the api you can use the app.

```
from pypads.app.base import PyPads
tracker = PyPads(autostart=True)
tracker.api.set_tag("foo", "bar")
```

Validators

Validators are to be used if the experimental status or code has to be checked on some properties. These should normally not log anything, but a validation report. A validation report should be an optional tag or at max a text file. In general validators should inform the user on runtime about errors and problems. It is planned to add the possibility to interrupt an execution if validators fail in the future. Some validators will be logging functions bound to library functions. An examplary validator which will want to be bound to the usage of pytorch is the determinism check for pytorch.

```
@validator
def determinism(self):
    check_determinism()
```

To call the api you can use the app.

```
from pypads.app.base import PyPads
tracker = PyPads(autostart=True)
tracker.validators.set_tag("foo", "bar")
```

Setup / Teardown functions

Setup or teardown functions are to be called when a run starts or ends. These mostly are used to log meta information about the experiment including data about git, hardware and the environment. A list of currently defined decorators can be found here.

```
class ICpu(PreRunFunction):

@staticmethod
def _needed_packages():
    return ["psutil"]

def _call(self, pads, *args, **kwargs):
    import psutil
    pads.api.set_tag("pypads.system.cpu.physical_cores", psutil.cpu_
↪count(logical=False))
    pads.api.set_tag("pypads.system.cpu.total_cores", psutil.cpu_count(logical=True))
    freq = psutil.cpu_freq()
    pads.api.set_tag("pypads.system.cpu.max_freq", f"{freq.max:2f}Mhz")
    pads.api.set_tag("pypads.system.cpu.min_freq", f"{freq.min:2f}Mhz")
```

Configuring setup or teardown functions can be done via the app constructor or api.

```
from pypads.app.base import PyPads
tracker = PyPads(setup_fns=[ICpu()], autostart=True)
# tracker.api.register_setup("custom_cpu", ICpu())
```

MappingFiles

Mapping files deliver hooks into libraries to trigger tracking functionality. They are written in yml and defining a syntax to markup functions, classes and modules.

Decorators

Decorators can be used instead of a mapping file to denote hooks in code. Because most libraries are not to be changed directly they are currently used sparingly. In PyPads defined decorators can be found here.

Logging functions

Logging functions are the generic functions performing tracking tasks bound to hooked functions of libraries. Everything not fitting into other concepts is just called logging function. Following function would track the input to the hooked function.

```
class Input(LoggingFunction):
"""
Function logging the input parameters of the current pipeline object function call.
"""

def __pre__(self, ctx, *args, _pypads_write_format=WriteFormats.pickle, _pypads_env:␣
↪LoggingEnv, **kwargs):
    """
    :param ctx:
    :param args:
    :param _pypads_write_format:
    :param kwargs:
    :return:
    """
    for i in range(len(args)):
        arg = args[i]
```

(continues on next page)

(continued from previous page)

```
        name = os.path.join(_pypads_env.call.to_folder(),
                            "args",
                            str(i) + "_" + str(id(_pypads_env.callback)))
        try_write_artifact(name, arg, _pypads_write_format)

    for (k, v) in kwargs.items():
        name = os.path.join(_pypads_env.call.to_folder(),
                            "kwargs",
                            str(k) + "_" + str(id(_pypads_env.callback)))
        try_write_artifact(name, v, _pypads_write_format)
```

Configuring logging functions can be achieved by providing mappings to the constructor of the app. Mapping files provide hooks (generally prepended by "pypads" in their naming) and logging functions are mapped to events. A hook can subsequently trigger multiple events and thus logging functions. To pass an event to function mapping a simple dict can be used.

```
from pypads.app.base import PyPads
event_function_mapping = {
    "parameters": Parameters(),
    "output": Output(_pypads_write_format=WriteFormats.text.name),
    "input": Input(_pypads_write_format=WriteFormats.text.name)
}
tracker = PyPads(events=event_function_mapping, autostart=True)
```

Additionally a hook to event mapping can be defined.

```
from pypads.app.base import PyPads
hook_event_mapping = {
    "parameters": {"on": ["pypads_fit"]},
    "output": {"on": ["pypads_fit", "pypads_predict"]},
    "input": {"on": ["pypads_fit"], "with": {"_pypads_write_format": WriteFormats.
↪text.name}},
}
tracker = PyPads(hooks=hook_event_mapping, autostart=True)
```

Defining hooks can be done via api, mappings, mapping files or decorators. Decorators are a sensible approach for local custom code.

```
from pypads.app.base import PyPads
tracker = PyPads(autostart=True)

@tracker.decorator.track(event="pypads_fit")
def fit_function_to_track(foo: str):
    return foo + "bar"
```

The same holds true for api based tracking.

```
from pypads.app.base import PyPads
tracker = PyPads(autostart=True)

def fit_function_to_track(foo: str):
    return foo + "bar"

tracker.api.track(ctx=get_class_that_defined_method(fit_function_to_track), fn=fit_
↪function_to_track, hooks=["pypads_fit"])
```

Mapping files or mappings are a more permanent, shareable and modular approach.

```
from pypads.app.base import PyPads
serialized_mapping = """
    metadata:
      author: "Thomas Weißgerber"
      version: "0.0.1"
      library:
        name: "test_foo"
        version: "0.1"

    mappings:
        :my_package.my_class.fit_function_to_track:
                events: "pypads_fit"
    """

tracker = PyPads(mapping=SerializedMapping("test_foo", serialized_mapping),␣
↪autostart=True)

def fit_function_to_track(foo: str):
    return foo + "bar"
```

Check points

Check points are currently not implemented. They will introduce a structured way to denote cache able states. By defining check points we hope to be able to define marks from which an experiment can be rerun in the future.

2.1.3 Examples

Sklearn DecisionTree example

Following shows how PyPads can be used to track the parameters, input and output of a sklearn experiment.

```
# define the configuration, in this case we want to track the parameters,
# outputs and the inputs of each called function included in the hooks (pypads_fit,␣
↪pypads_predict)
events = {
    "parameters": {"on": ["pypads_fit"]},
    "output": {"on": ["pypads_fit", "pypads_predict"]},
    "input": {"on": ["pypads_fit"]}
}
# A simple initialization of the class will activate the tracking
PyPads(events=events)

# An example
from sklearn import datasets, metrics
from sklearn.tree import DecisionTreeClassifier

# load the iris datasets
dataset = datasets.load_iris()

# fit a model to the data
model = DecisionTreeClassifier()
model.fit(dataset.data, dataset.target) # pypads will track the parameters, output,␣
↪and input of the model fit function.
# get the predictions
```

(continues on next page)

(continued from previous page)

```
predicted = model.predict(dataset.data) # pypads will track only the␣
↪model predict function.
```

The used hooks for each event are defined in the mapping yml file where each event includes the functions to listen to.

Mapping file example

For the previous example, the sklearn mapping yml file would look like the following.

```
metadata:
  author: "Thomas Weißgerber"
  version: "0.1.0"
  library:
    name: "sklearn"
    version: "0.19.1"

mappings:
  !!python/pPath sklearn:
    !!python/pPath base.BaseEstimator:
      data:
        concepts: ["algorithm"]
    !!python/pPath __init__:
      hooks: "pypads_init"
    !!python/rSeg (fit|.fit_predict|fit_transform)$:
      hooks: "pypads_fit"
    !!python/rSeg (fit_predict|predict|score)$:
      hooks: "pypads_predict"
    !!python/rSeg (fit_transform|transform)$:
      hooks: "pypads_transform"
```

For example, “pypads_fit” is an event listener on any **fit, fit_predict and fit_transform** function call made by any tracked class with those methods.

Defining a hook for an event

A hook can be defined in the mapping file via the “hooks” attribute. It is composed of the given name and path defined by the keys in the yml file. Muliple hooks can use the same name and therefore trigger the same functions.

Define an event

Once the hooks are defined, they are then linked to the events we want them to trigger. Following the example below, the hook **pypads_metric** will be linked to an event we call **Metrics** for example. This is done via passing a dictionary as the parameter **config** to the *PyPads class*

```
events = {
            "Metrics" : {"on": ["pypads_metrics"]}
         }
```

PyPads loggers

PyPads has a set of built-in logging functions that are mapped by default to some pre-defined events. Check the default setting of PyPads *here*. The user can also define custom logging functions for custom events. Details on how to do that can be found *here*.

2.1.4 External resources

Currently there are unfortunately not too many external resources available fo PyPads. Additional examples are to be added in the next steps of the road map. You can find an IPython Notebook and an Code example on these repositories.

TODO Please add links to two repositories with example code (We can use the stuff for the data science lab)

CHAPTER 3

PyPads

3.1 PyPads application class

This class represents the app. To start and activate the tracking of your modules, classes and functions, the app class has to be instantiated.

> **Warning:** It is recommended to initialize the tracking **before** importing the modules to be tracked. While extending the importlib and reloading the modules may work sometimes doing so may result in unforeseen issues.

3.1.1 Default settings

The app includes default values for configuration, hook/event mappings, event/function mappings etc.

Default Anchors

Anchors are names for repeating types of hooks. Fit functions for example are existing on multiple libraries.

```
DEFAULT_ANCHORS = [Anchor("pypads_init", "Used if a tracked concept is initialized."),
                   Anchor("pypads_fit", "Used if an model is fitted to data."),
                   Anchor("pypads_predict", "Used if an model predicts something."),
                   Anchor("pypads_metric", "Used if an metric is compiled."),
                   Anchor("pypads_log", "Used to only log a call.")]
```

Default Event Types

Event types represent strategies to react to an anchor / hook.

```
DEFAULT_EVENT_TYPES = [EventType("parameters", "Track the parameters for given model.
↪"),
                       EventType("output", "Track the output of the function."),
                       EventType("input", "Track the input of the function."),
                       EventType("hardware", "Track current hardware load on function␣
↪execution."),
                       EventType("metric", "Track a metric."),
                       EventType("autolog", "Activate mlflow autologging."),
                       EventType("pipeline", "Track a pipeline step."),
                       EventType("log", "Log the call to console."),
                       EventType("init", "Log the tracked class init to console.")]
```

Default Config

The configuration for PyPads

```
DEFAULT_CONFIG = {
    "track_sub_processes": False,  # Activate to track spawned subprocesses by␣
↪extending the joblib
    "recursion_identity": False, # Activate to ignore tracking on recursive calls of␣
↪the same function with the same mapping
    "recursion_depth": -1,  # Limit the tracking of recursive calls
    "log_on_failure": True,  # Log the stdout / stderr output when the execution of␣
↪the experiment failed
    "include_default_mappings": True  # Include the default mappings additionally to␣
↪the passed mapping if a mapping is passed
}
```

Default Hook Mapping

The hook mapping, maps hooks (anchors) to the events (event types).

```
DEFAULT_HOOK_MAPPING = {
    "init": {"on": ["pypads_init"]},
    "parameters": {"on": ["pypads_fit"]},
    "hardware": {"on": ["pypads_fit"]},
    "output": {"on": ["pypads_fit", "pypads_predict"]},
    "input": {"on": ["pypads_fit"], "with": {"_pypads_write_format": WriteFormats.
↪text.name}},
    "metric": {"on": ["pypads_metric"]},
    "pipeline": {"on": ["pypads_fit", "pypads_predict", "pypads_transform", "pypads_
↪metric"]},
    "log": {"on": ["pypads_log"]}
}
```

Default Event Mapping

Defines which logging functions should be run for events.

```
DEFAULT_LOGGING_FNS = {
    "parameters": Parameters(),
    "output": Output(_pypads_write_format=WriteFormats.text.name),
    "input": Input(_pypads_write_format=WriteFormats.text.name),
```

(continues on next page)

(continued from previous page)

```
    "hardware": [Cpu(), Ram(), Disk()],
    "metric": Metric(),
    "autolog": MlflowAutologger(),
    "pipeline": PipelineTracker(_pypads_pipeline_type="normal", _pypads_pipeline_
↪args=False),
    "log": Log(),
    "init": LogInit()
}
```

3.2 Logging functions

3.2.1 LoggingFunction base class

To develop custom loggers, we need to write a class that inherits from the base class **LoggingFunction**. Then, those custom loggers can be mapped to events of the user choice in the parameter **mapping** of the *PyPads class*.

3.2.2 Pre and Post run loggers

Another type of logging functions supported by Pypads is the pre/post run loggers which are executed before and after the run execution respectively.

- Pre Run loggers

class pypads.app.injections.run_loggers.**PreRunFunction**(**args*, ***kwargs*)

This class should be used to define new pre run functions

_call(*pads*, **args*, ***kwargs*)

Function where to add you custom code to execute before starting or ending the run.

Parameters **pads** – the current instance of PyPads.

_check_dependencies()

Raise error if dependencies are missing.

static _needed_packages()

Overwrite this to provide your package names. :return: List of needed packages by the logger.

- Post Run loggers

class pypads.app.injections.run_loggers.**PostRunFunction**(**args*, ***kwargs*)

This class should be used to define new post run functions

_call(*pads*, **args*, ***kwargs*)

Function where to add you custom code to execute before starting or ending the run.

Parameters **pads** – the current instance of PyPads.

_check_dependencies()

Raise error if dependencies are missing.

static _needed_packages()

Overwrite this to provide your package names. :return: List of needed packages by the logger.

3.2.3 Mlflow autolog (experimental)

Pypads also support mlflow autologging functionalities. More on that can be found at MLflow.

class pypads.injections.loggers.mlflow.mlflow_autolog.**MlflowAutologger**(**args, order=-1, **kwargs*)

MlflowAutologger is the intergration of the mlflow autologging functionalities into PyPads tracking system.

__call_wrapped__(*ctx, *args, _args, _kwargs, _pypads_autologgers=None, _pypads_env=<class 'pypads.injections.analysis.call_tracker.LoggingEnv'>, **kwargs*)

Note: Experimental: This method may change or be removed in a future release without warning.

Function used to enable autologgers of mlflow.

identity

Return the identity of the logger. This should be unique for the same functionality across multiple versions. :return:

3.3 Utilities

pypads.utils.util.**dict_merge**(**dicts*)

Simple merge of dicts :param dicts: :return:

pypads.utils.util.**dict_merge_caches**(**dicts*)

Merge two dicts. Entries are overwritten if not mergeable. Cache is supported. :param dicts: dicts to merge :return:

pypads.utils.util.**get_class_that_defined_method**(*meth*)

Try to find the class / module which defined given method. :param meth: Method for which we search an origin. :return:

pypads.utils.util.**inheritors**(*clazz*)

Function getting all subclasses of given class. :param clazz: Clazz to search for :return:

pypads.utils.util.**is_package_available**(*name*)

Check if given package is available. :param name: Name of the package :return:

pypads.utils.util.**local_uri_to_path**(*uri*)

Convert URI to local filesystem path.

pypads.utils.util.**sizeof_fmt**(*num, suffix='B'*)

Get the mem / disk size in a human readable way. :param num: :param suffix: :return:

pypads.utils.util.**string_to_int**(*s*)

Build a int from a given string. :param s: :return:

3.4 Mapping Files

PyPads using the concept of mapping files to track which functions should be logged. These files are written in YAML. YAML (YAML Ain't A Markup Language) is a human readable data serialization language. YAML has features such as comments and anchors these features make it desirable.

The mapping file can be divided broadly into different parts like metadata, fragments and mappings. Each section is explained in detail below. Following excerpts show possible mapping files. While the keras file uses implicit syntax for the path matchers marked by a prepending `:`, the sklearn version depicts how to use YAML typing with `!!python/pPath`, `!!python/rSeg` or `!!python/pSeg`.

```yaml
metadata:
  author: "Thomas Weißgerber"
  version: "0.1.0"
  library:
    name: "keras"
    version: "2.3.1"

mappings:
  :keras.metrics.Metric:
    hooks: ["pypads_metric"]
    data:
      concepts: ["keras classification metrics"]

  :keras.engine.training.Model:
    :__init__:
      hooks: ["pypads_init"]
    :{re:(fit|fit_generator)$}:
      hooks: ["pypads_fit"]
    :predict_classes:
      hooks: ["pypads_predict"]
```

```yaml
metadata:
  author: "Thomas Weißgerber"
  version: "0.1.0"
  library:
    name: "sklearn"
    version: ">= 0.19.1"

fragments:
  default_model:
    !!python/pPath __init__:
      hooks: "pypads_init"
    !!python/rSeg (fit|.fit_predict|fit_transform)$:
      hooks: "pypads_fit"
    !!python/rSeg (fit_predict|predict|score)$:
      hooks: "pypads_predict"
    !!python/rSeg (fit_transform|transform)$:
      hooks: "pypads_transform"

mappings:
  !!python/pPath sklearn:
    !!python/pPath base.BaseEstimator:
      ;default_model: ~
      data:
        concepts: ["algorithm"]
    !!python/pPath metrics.classification:
      !!python/rSeg .*:
        hooks: "pypads_metric"
        data:
          concepts: ["Sklearn provided metric"]
    !!python/pPath tree.tree.DecisionTreeClassifier:
      ;default_model: ~
```

(continues on next page)

(continued from previous page)

```
      data:
        name: decision tree classifier
        other_names: []
        type: Classification
        hyper_parameters:
          model_parameters:
            - name: split_quality
              kind_of_value: "{'gini', 'entropy'}"
              optional: 'True'
              description: The function to measure the quality of a split.
              default_value: "'gini'"
              path: criterion
            - name: splitting_strategy
              kind_of_value: "{'best', 'random'}"
              optional: 'True'
              description: The strategy used to choose the split at each node.
              default_value: "'best'"
              path: splitter
            - name: max_depth_tree
              kind_of_value: integer
              optional: 'True'
              description: The maximum depth of the tree.
              default_value: None
              path: max_depth
            - name: min_samples_split
              kind_of_value: "{integer, float}"
              optional: 'True'
              description: The minimum number of samples required to split an␣
↪internal node.
              default_value: '2'
              path: min_samples_split
            - name: min_samples_leaf
              kind_of_value: "{integer, float}"
              optional: 'True'
              description: The minimum number of samples required to be at a leaf␣
↪node.
              default_value: '1'
              path: min_samples_leaf
            - name: min_weight_fraction_leaf
              kind_of_value: float
              optional: 'True'
              description: The minimum weighted fraction of the sum total of weights␣
↪(of all
                the input samples) required to be at a leaf node.
              default_value: '1'
              path: min_weight_fraction_leaf
            - name: max_features
              kind_of_value: "{integer, float, 'auto', 'sqrt', 'log2', None}"
              optional: 'True'
              description: The number of features to consider when looking for the␣
↪best split.
              default_value: None
              path: max_features
            - name: random_state
              kind_of_value: "{integer, RandomState instance, None}"
              optional: 'True'
              description: The seed of the pseudo random number generator to use when␣
↪shuffling
```

(continues on next page)

(continued from previous page)

```
                the data. If int, random_state is the seed used by the random number␣
↪generator;
                If RandomState instance, random_state is the random number generator;␣
↪If None,
                the random number generator is the RandomState instance used by np.
↪random.
              default_value: None
              path: random_state
            - name: max_leaf_nodes
              kind_of_value: integer
              optional: 'True'
              description: Grow a tree with max_leaf_nodes in best-first fashion.
              default_value: None
              path: max_leaf_nodes
            - name: min_impurity_decrease
              kind_of_value: float
              optional: 'True'
              description: A node will be split if this split induces a decrease of␣
↪the impurity
                greater than or equal to this value.
              default_value: '0'
              path: min_impurity_decrease
            - name: class_weight
              kind_of_value: "{dict, list of dicts, 'balanced', None}"
              optional: 'False'
              description: Weights associated with classes.
              default_value: None
              path: class_weight
          optimisation_parameters:
            - name: presort
              kind_of_value: "{boolean, 'auto'}"
              optional: 'True'
              description: Whether to presort the data to speed up the finding of␣
↪best splits
                in fitting.
              default_value: "'auto'"
              path: presort
          execution_parameters: []
```

Metadata The metadata part contains information about the author, the mapping file version and the library information. The mapping file version is required so that a change in the tracking functionalities can be easily traced to the version of the mapping file. Even while having the same library version, a user can modify the mapping file to track additional functions of the library or remove some tracking functionalities. Such changes need to be handled to provide better experiment tracking and reproducibility. PyPads does this via versioning of the mapping file. Another tag called "library" contains information about the library which the mapping file addresses such as the name of the library and the version of the library. This metadata section helps PyPads track different versions of libraries without them having a conflict.

```
metadata:
  author: "Thomas Weißgerber"
  version: "0.1.0"
  library:
    name: "sklearn"
    version: "0.19.1"
```

Fragments Repeated patterns in the library can be included in the fragments section of the mappings file. Fragments allows users to link functions across classes. For example, in scikit-learn the fit function is a function for fitting the estimators. All classification/regression estimators will have a fit function. In such a scenario, the user does not have to write mappings for each and every estimator. Instead, the user can add the function to the fragments part and PyPads will automatically log those functions.

```
fragments:
  default_model:
    .__init__:
      events: "pypads_init"
    .{re:(fit|.fit_predict|fit_transform)$}:
      events: "pypads_fit"
```

Mappings This part in the mapping file gives information to PyPads about the functions to track. In the example, we use the sklearn base estimator to encompass all logging functionalities from a single point. The user can add other classes as shown with the Decision Tree Classifier. By doing this the user also has to provide all the hyperparameters so that PyPads knows what to track. For each hyperparameter the user also has to provide the name of the hyperparameter, whether it is optional or not, its description and so forth.

3.4.1 Concepts

PyPads mapping files contain keys called concepts. When creating a main key in the mappings file, it could be anything such as a metric, a dataset, splitting strategy, an algorithm and so forth. The concepts key present within the main key links the main key to previously determined categories such as metric, dataset or algorithm to name a few. This helps PyPads recognize what type the main key is and how to process it.

3.4.2 Notations

PyPads can accept different notations through the YAML parser. Users can use the power of regular expressions to specify function groups that should trigger specific events. Here in the below given example, we hook all functions in sklearn.metrics.classification to “pypads_metric”. We also inform PyPads that all functions of this form are an instance of sklearn provided metrics using the concepts key.

```
mappings:
  sklearn:
    .metrics.classification.{re:.*}:
       data:
         concepts: ["Sklearn provided metric"]
         events: "pypads_metric"
```

3.4.3 Adding a new mapping file

When a user wants to add their own mapping file, they have to follow the following steps # Create a YAML mapping file in the path pypads/bindings/resources/mapping with the appropriate name and version number # Add a metadata part containing information about the author, version of the mapping file and library # Add fragments if a general function name is present. You can use regex to specify the patterns # Add mappings for metrics, datasets etc is they are present # PyPads will pick up the information when it is restarted.

CHAPTER 4

Extensions

- **PaDRe-Pads** is a tool that builds on PyPads and add some semantics to the tracked data of Machine learning experiments. See the padre-pads documentation.

4.1 Extensions and plugins

PyPads features a plugin system to extend its functionality. Currently following plugins are being developed.

pypads_padre Also called PadrePads introduces additional concepts of machine learning. While PyPads is fairly unopinionated about what it is logging PadrePads tries to impose some structure.

pypads_onto (unreleased) Also called OntoPads introduces ontology mappings to pypads. It is based on the other plugin PadrePads and will enable given concept unique references.

4.1.1 PadrePads

PadrePads builds upon pypads when it comes to tracking, but it also adds a layer of loggers that tracks semantic information from experiments executions. The documentation for PadrePads can be found here.

CHAPTER 5

Related Projects

- **PyPadre** is the predecessor of PadrePads. Its development has been discontinued.

5.1 Related Projects

PyPads was developed of some related projects. One of which was its predecessor PyPadre.

5.1.1 PyPaDRE

PyPaDRE was the original experimental project developed on the context of padre-lab to track machine learning experiments. After identifying drawbacks in the basic architecture or pypadre the development on pypads was started. The documentation for PyPaDRE can be found here.

CHAPTER 6

About Us

This work has been developed within the **Data Science Chair** of the University of Passau. It has been partially funded by the **Bavarian Ministry of Economic Affairs, Regional Development and Energy** by means of the funding programm “**Internetkompetenzzentrum Ostbayern**” as well as by the **German Federal Ministry of Education and Research** in the project “**Provenance Analytics**” with grant agreement number 03PSIPT5C.

6.1 About Us

This work has been developed within the **Data Science Chair** of the University of Passau. It has been partially funded by the **Bavarian Ministry of Economic Affairs, Regional Development and Energy** by means of the funding programm “**Internetkompetenzzentrum Ostbayern**” as well as by the **German Federal Ministry of Education and Research** in the project “**Provenance Analytics**” with grant agreement number 03PSIPT5C.

Python Module Index

Index