PyPads

Release 0.1.20

May 19, 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!

2.1.1 Usage

pypads is easy to use. Just define what is needed to be tracked in the config and call PyPads.

A simple example looks like the following.:

```
from pypads.base import PyPads
# 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)
config = {"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(config=config)
```

(continues on next page)

(continued from previous page)

```
# 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.
```

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

2.1.2 Mapping file example

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

```
{
  "default_hooks": {
    "modules": {
      "fns": {}
    },
    "classes": {
      "fns": {
        "pypads_init": [
          "__init__"
        ],
        "pypads_fit": [
          "fit",
          "fit_predict",
          "fit_transform"
        ],
        "pypads_predict": [
          "fit_predict",
          "predict",
          "score"
        ],
        "pypads_transform": [
          "fit_transform",
          "transform"
        ]
      }
    },
    "fns": {}
  },
  "algorithms": [
    {
      "name": "base sklearn estimator",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.base.BaseEstimator"
```

(continues on next page)

(continued from previous page)

```
    },
    "hooks": {
      "pypads_fit": [
        "fit",
        "fit_predict",
        "fit_transform"
      ],
      "pypads_predict": [
        "fit_predict",
        "predict"
      ],
      "pypads_transform": [
        "fit_transform",
        "transform"
      ]
    }
  },
  {
    "name": "sklearn classification metrics",
    "other_names": [],
    "implementation": {
      "sklearn": "sklearn.metrics.classification"
    },
    "hooks": {
      "pypads_metric": [
        ".*"
      ]
    }
  },
  {
    "name": "sklearn datasets",
    "other_names": [],
    "implementation": {
      "sklearn": "sklearn.datasets"
    },
    "hooks": {
      "pypads_dataset": [
        "load*"
      ]
    }
  },
  {
    "name": "sklearn cross validation",
    "other_names": [],
    "implementation": {
      "sklearn": "sklearn.model_selection._search.BaseSearchCV"
    },
    "hooks": {
      "pypads_param_search": [
        "fit"
      ]
    }
  },
  {
    "name": "sklearn cross validation",
    "other_names": [],
    "implementation": {
```

(continues on next page)

(continued from previous page)

```
        "sklearn": "sklearn.model_selection._validation._fit_and_score"
      },
      "hooks": {
        "pypads_param_search_exec": "always"
      }
    },
    {
      "name": "sklearn cross validation",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.model_selection._split.BaseCrossValidator"
      },
      "hooks": {
        "pypads_split": [
          "split"
        ]
      }
    },
    {
      "name": "sklearn shuffle split",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.model_selection._split.BaseShuffleSplit"
      },
      "hooks": {
        "pypads_split": [
          "split"
        ]
      }
    },
    {
      "name": "base sklearn estimator",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.base.RegressorMixin"
      },
      "hooks": {
        "pypads_metric": [
          "score"
        ]
      }
    },
    {
      "name": "base sklearn estimator",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.base.ClassifierMixin"
      },
      "hooks": {
        "pypads_metric": [
          "score"
        ]
      }
    },
    {
      "name": "base sklearn estimator",
      "other_names": [],
```

(continues on next page)

(continued from previous page)


```
      "implementation": {
        "sklearn": "sklearn.base.DensityMixin"
      },
      "hooks": {
        "pypads_metric": [
          "score"
        ]
      }
    },
    {
      "name": "base decision tree",
      "other_names": [],
      "implementation": {
        "sklearn": "sklearn.tree.tree.BaseDecisionTree"
      }
    },
    {
      "name": "logistic regression",
      "other_names": [
        "logit regression",
        "maximum-entropy classification",
        "MaxEnt",
        "log-linear classifier"
      ],
      "implementation": {
        "sklearn": "sklearn.linear_model.logistic.LogisticRegression"
      },
      "type": "Classification",
      "hyper_parameters": {
        "model_parameters": [
          {
            "name": "penalty_norm",
            "kind_of_value": "{l1, l2}",
            "optional": "False",
            "description": "Used to specify the norm used in the penalization.",
            "sklearn": {
              "default_value": "'l2'",
              "path": "penalty"
            }
          },
          {
            "name": "dual",
            "kind_of_value": "boolean",
            "optional": "False",
            "description": "Dual or primal formulation.",
            "sklearn": {
              "default_value": "False",
              "path": "dual"
            }
          },
          {
            "name": "tolerance",
            "kind_of_value": "float",
            "optional": "False",
            "description": "Tolerance for stopping criteria.",
            "sklearn": {
              "default_value": "0.0001",
```


(continues on next page)

(continued from previous page)


```
              "path": "tol"
            }
          },
          {
            "name": "inverse_regularisation_strength",
            "kind_of_value": "float",
            "optional": "False",
            "description": "Inverse of regularization strength; must be a positive␣
↪float. Like in support vector machines, smaller values specify stronger␣
↪regularization.",
            "sklearn": {
              "default_value": "1.0",
              "path": "C"
            }
          },
          {
            "name": "fit_intercept",
            "kind_of_value": "boolean",
            "optional": "False",
            "description": "Specifies if a constant (a.k.a. bias or intercept) should␣
↪be added to the decision function.",
            "sklearn": {
              "default_value": "True",
              "path": "fit_intercept"
            }
          },
          {
            "name": "intercept_scaling",
            "kind_of_value": "float",
            "optional": "False",
            "description": "Useful only when the solver \\u2018liblinear\\u2019 is␣
↪used and self.fit_intercept is set to True. In this case, x becomes [x, self.
↪intercept_scaling], i.e. a \\u201csynthetic\\u201d feature with constant value␣
↪equal to intercept_scaling is appended to the instance vector. The intercept␣
↪becomes intercept_scaling * synthetic_feature_weight.",
            "sklearn": {
              "default_value": "1",
              "path": "intercept_scaling"
            }
          },
          {
            "name": "class_weight",
            "kind_of_value": "{dict, 'balanced', None}",
            "optional": "False",
            "description": "Weights associated with classes.",
            "sklearn": {
              "default_value": "None",
              "path": "class_weight"
            }
          },
          {
            "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 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.",
```


(continues on next page)

(continued from previous page)


```
            "sklearn": {
              "default_value": "None",
              "path": "random_state"
            }
          },
          {
            "name": "solver",
            "kind_of_value": "{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}",
            "optional": "False",
            "description": "Solver to use in the computational routines.",
            "sklearn": {
              "default_value": "'liblinear'",
              "path": "solver"
            }
          },
          {
            "name": "multi_class",
            "kind_of_value": "{'ovr', 'multinomial'}",
            "optional": "False",
            "description": "If the option chosen is \\u2018ovr\\u2019, then a binary␣
↪problem is fit for each label. Else the loss minimised is the multinomial loss fit␣
↪across the entire probability distribution.",
            "sklearn": {
              "default_value": "'ovr'",
              "path": "multi_class"
            }
          },
          {
            "name": "verbose",
            "kind_of_value": "integer",
            "optional": "True",
            "description": "For the liblinear and lbfgs solvers set verbose to any␣
↪positive number for verbosity.",
            "sklearn": {
              "default_value": "0",
              "path": "verbose"
            }
          }
        ],
        "optimisation_parameters": [
          {
            "name": "max_iterations",
            "kind_of_value": "integer",
            "optional": "False",
            "description": "Maximum number of iterations.",
            "sklearn": {
              "default_value": "100",
              "path": "max_iter"
            }
          },
          {
            "name": "reuse_previous",
            "kind_of_value": "boolean",
            "optional": "False",
            "description": "When set to True, reuse the solution of the previous call␣
↪to fit as initialization, otherwise, just erase the previous solution.",
            "sklearn": {
```


(continues on next page)

(continued from previous page)

```
            "default_value": "False",
            "path": "warm_start"
          }
        },
        {
          "name": "jobs",
          "kind_of_value": "integer",
          "optional": "False",
          "description": "Number of CPU cores used when parallelizing over classes.
↪",
          "sklearn": {
            "default_value": "1",
            "path": "n_jobs"
          }
        }
      ],
      "execution_parameters": []
    }
  }
 ],
 "metadata": {
   "author": "Michael Granitzer",
   "library": "sklearn",
   "library_version": "0.19.1",
   "mapping_version": "0.1"
 }
}
```

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.

2.1.3 Defining a hook for an event

A hook can be defined in the mapping file with 3 different ways.

1. Always:

```
{
  "name": "sklearn classification metrics",
  "other_names": [],
  "implementation": {
    "sklearn": "sklearn.metrics.classification"
  },
  "hooks": {
    "pypads_metric": "always"
  }
}
```

   This hook triggers always. If you annotate a module with this hook, all its functions and classes will be tracked.

2. QualNameHook:

```
{
  "name": "sklearn classification metrics",
  "other_names": [],
  "implementation": {
```

(continues on next page)

(continued from previous page)

```
    "sklearn": "sklearn.metrics.classification"
  },
  "hooks": {
    "pypads_metric": ["f1_score"]
  }
}
```

Tracks function with a name matching the given Regex.

3. PackageNameHook:

```
{
  "name": "sklearn classification metrics",
  "other_names": [],
  "implementation": {
    "sklearn": "sklearn.metrics"
  },
  "hooks": {
    "pypads_metric": [{"type": "package_name", "name":".*classification.*"}]
  }
}
```

Tracks all attributes of the module where "package_name" is matching Regex.

2.1.4 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*:

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

2.1.5 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*).

CHAPTER 3

PyPads

3.1 PyPads

3.1.1 PyPads Class

To start and activate the tracking of your modules, classes and functions, An instantiation of the base class has to be done.

It is recommended to initialize the tracking **before** importing the modules to be tracked.

class pypads.base.**PyPads**(*uri=None*, *folder=None*, *name=None*, *mapping_paths=None*, *mapping=None*, *init_run_fns=None*, *include_default_mappings=True*, *logging_fns=None*, *config=None*, *reload_modules=False*, *reload_warnings=True*, *clear_imports=False*, *affected_modules=None*, *pre_initialized_cache=None*, *disable_run_init=True*)

PyPads app and base class. It enable automatic logging for all libraries included in the mapping files. Serves as the main entrypoint to PyPads. After constructing this app tracking is activated.

param uri string, optional (default=None) Address of local or remote tracking server that **MLflow** uses to record runs. If None, then it tries to get the environment variable **'MLFLOW_PATH'** or the **'HOMEPATH'** of the user.

param name string, optional (default=None) Name of the **MLflow** experiment to track.

param mapping_paths list, optional (default=None) Absolute paths to additional mapping files.

param mapping dict, optional (default=None) Mapping to the logging functions to use for the tracking of the events. If None, then a DEFAULT_MAPPING is used which allow to log parameters, outputs or inputs.

param init_run_fns list, optional (default=None) Logging function to execute on tracking initialization.

param include_default_mappings boolean, optional (default=True) A flag whether to use the default provided mappings or not.

param logging_fns dict, optional (default=None) User defined logging functions to use where each dict item has to be ‘ “event”: fn’ or ‘ “event”: {fn1,fn2,. . . }’.

param config dict, optional (default=None) A dictionary that maps the events defined in PyPads mapping files with the logging functions.

param reload_modules boolean, optional (default=False) Reload and duck punch already loaded modules before the tracking activation if set to True.

param clear_imports boolean, optional (default=False) Delete alredy loaded modules for sys.modules() if set to True.

activate_tracking(*reload_modules=False*, *reload_warnings=True*, *clear_imports=False*, *affected_modules=None*)

Function to duck punch all objects defined in the mapping files. This should at best be called before importing any libraries. :param mod_globals: globals() object used to duckpunch already loaded classes :return:

add_atexit_fn(*fn*)

Add function to be executed before stopping your process.

Default settings

The default configuration of events/hooks:

```
DEFAULT_CONFIG = {"events": {
"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"]}
},
    "recursion_identity": False,
    "recursion_depth": -1,
    "log_on_failure": True}
```

The default mapping of events/loggers:

```
DEFAULT_LOGGING_FNS = {
"parameters": Parameters(),
"output": Output(_pypads_write_format=WriteFormats.text.name),
"input": Input(_pypads_write_format=WriteFormats.text.name),
"hardware": {Cpu(), Ram(), Disk()},
"metric": Metric(),
"autolog": MlflowAutologger(),
"pipeline": PipelineTracker(_pypads_pipeline_type="normal", _pypads_pipeline_
↪args=False),
"log": Log(),
"init": LogInit()
```

3.1.2 Logging functions

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*.

class pypads.functions.loggers.base_logger.**LoggingFunction**(**args*, *static_parameters=None*, ***kwargs*)

This class should be used to define new custom loggers. The user has to define __pre__ and/or __post__ methods depending on the specific use case.

Parameters **static_parameters** – dict, optional, static parameters (if needed) to be used when logging.

Note: It is not recommended to change the __call_wrapped__ method, only if really needed.

__call_wrapped__(*ctx*, **args*, *_pypads_env: pypads.functions.analysis.call_tracker.LoggingEnv*, *_args*, *_kwargs*, ***_pypads_hook_params*)

The real call of the wrapped function. Be carefull when you change this. Exceptions here will not be catched automatically and might break your workflow. The returned value will be passed on to __post__ function.

Returns _pypads_result

__post__(*ctx*, **args*, *_pypads_env*, *_pypads_pre_return*, *_pypads_result*, *_args*, *_kwargs*, ***kwargs*)

The function to be called after executing the log anchor.

Parameters

- **_pypads_pre_return** – the value returned by __pre__.
- **_pypads_result** – the value returned by __call_wrapped__.

Returns the wrapped function return value

__pre__(*ctx*, **args*, *_pypads_env*, *_args*, *_kwargs*, ***kwargs*)

The function to be called before executing the log anchor. the value returned will be passed on to the __post__ function as **_pypads_pre_return**.

Returns _pypads_pre_return

_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.

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.functions.pre_run.pre_run.**PreRunFunction**(**args*, *fn=None*, ***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 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.functions.post_run.post_run.**PostRunFunction**(**args*, *fn=None*, *message=None*, ***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 after 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.

Mlflow autolog (experimental)

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

class pypads.functions.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.functions.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.

3.1.3 Utilities

CHAPTER 4

Related Projects

- **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 Related Projects

4.1.1 PadrePads

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.

4.1.2 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

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.

5.1 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.

Index

Symbols

A

L

M

P