PyPaDRE Documentation Documentation

Michael Granitzer

Oct 19, 2020

CONTENTS

CHAPTER

ONE

PYPADRE CONCEPT

1.1 Goals

PaDRE (Platform for mAchine learning and Data science REproducibility) provides a platform to collect, compare and analyze machine learning experiments in a systematic and open way. It builds on Open Science principles and strives for improving research and education processes in disciplines related to Machine Learning and Data Science.

PyPaDRE is the Python-centric environment to create and execute Machine Learning and Data Science experiments. It aims to be a **minimal-invasive environment** for evaluating Machine Learning and Data Science experiments in a **systematic and structured** way. While PyPaDRE can be used as standalone, local solution for running and comparing experiments, all results can - and in the spirit of Open Science should - be made available via the PaDRE platform.

1.2 Use Cases

1.2.1 Rapid Development Support

Padre's first use case is for rapid development by reducing scaffolding scientific experiments. Users should be given

- easy access to datasets
- convenience functions for setting up experiments
- running the experiments
- comparing and visualising results
- packaging and deploying the experiment

Every experiment run should be logged in order to keep the experiments traceable / comparable.

1.2.2 Hyperparameter Optimization

After setting up an initial prototype, the next step is to do some initial hyperparameter tests. Padre should support this via a few lines of code and make it easy to analyse the optimized hyperparameters / runs.

1.2.3 Large Scale Experimentation

Large scale experimentation aims at a later stage in the research cycle. After developing an initial prototype there is often the need to setup larger experiments and compare the different runs with each other, show error bars, test for statistical significance etc. Large scale experimentation aims to define an experiment and hyperparameters to be tested and to deploy the experiment runs over a larger cluster. Results could be checked online and compared online using web-based visualisation. Results can be converted in different formats (e.g. latex, markedown).

1.2.4 Visual Analysis and Collaborative Exploration

Experimental results should be visualised / analysed in a interactive frontend and potentially discussed with others.

1.2.5 Interactive Publishing and Versioned Result Publishing

Experimental results should be accessible online and embedded in online publications (e.g. Blogs). When new versions of experiments are available, a new version of the online publication should appear. The versions need to be trackable.

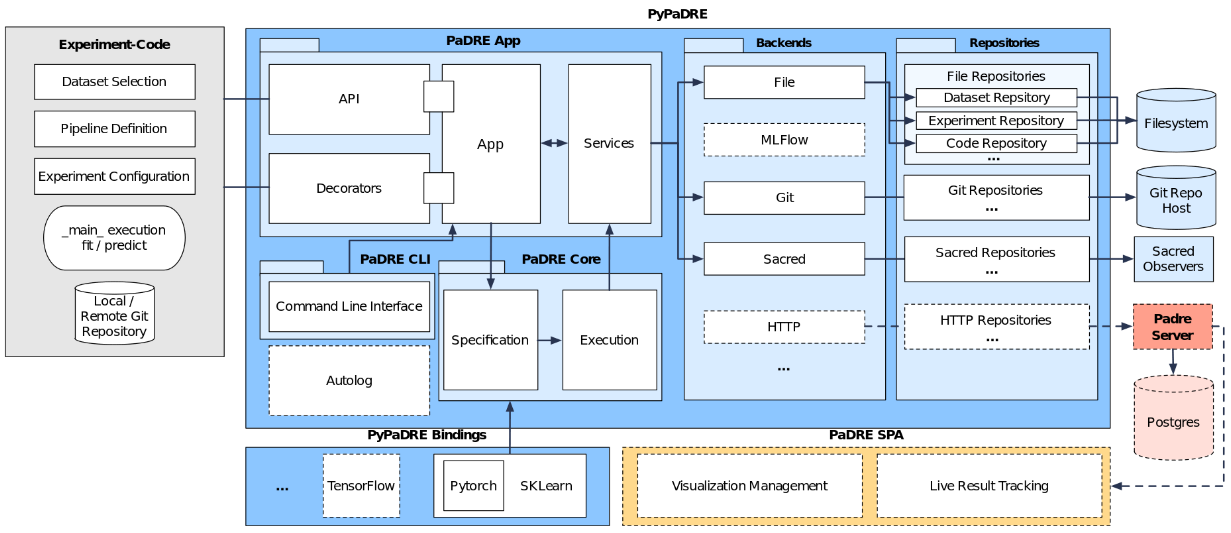
1.2.6 Research Management and Collaboration

Managing a smaller (PhD) or larger (funded project) research project should be fully supported by padre. This includes

- source code management
- experiment management and evaluation
- formulation of research questions
- collaboration on experiments, interpretation and writing
- issue tracking and discussion

1.3 Architecture

The following figure shows the overall architecture.



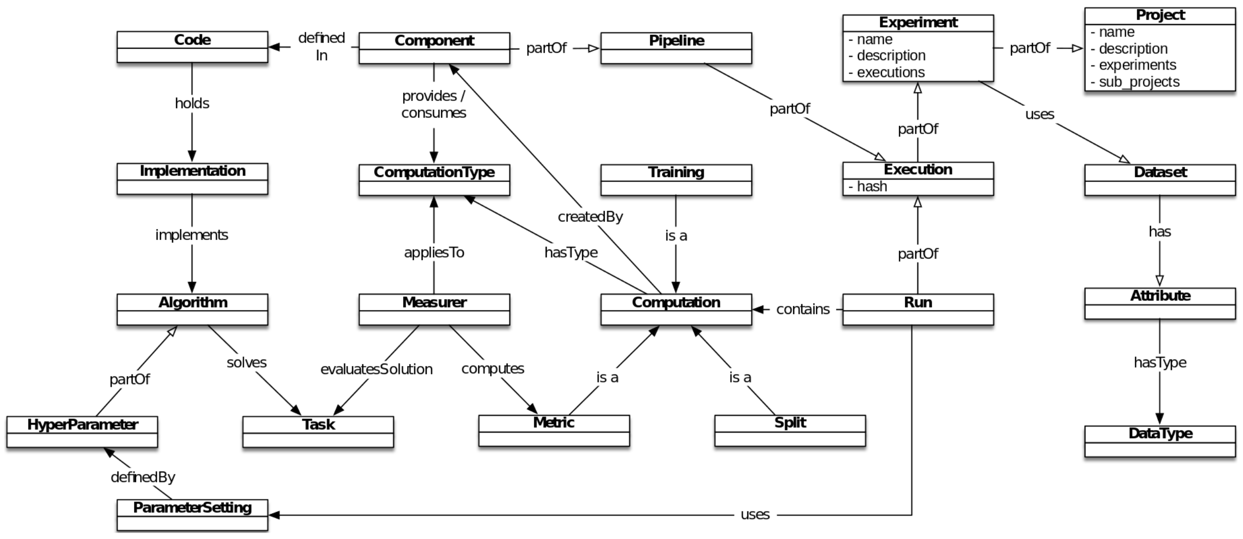
PyPaDRE provides means to

- Manage local and remote data sets

- Create, run and compare experiments
- Automatically identify hyperparameters and components of workflows from SKLearn and PyTorch
- Provide an easy to use Python and CLI interface
- Automatically version datasets, hyperparameters, code etc required for the experiment

while being minimally invasive.

Researchers should use their favorite tools and environments in order to conduct there research while PyPaDRE takes care of managing resources and experiments in the background.



This image shows the different parts of an experiment and how they all work together in the PaDRe architecture.

1.3.1 Experiments and Pipelines

In order to do so, PyPaDRE defines four core concepts: pipelines, experiments, runs and splits.

- **Pipelines/Workflows** are the actual machine learning workflows which can be trained and tested. The pipeline is composed of different components which are basically data processes that work on the input data. They can be preprocessing algorithms, splitting strategies, classification or regression algorithms. PaDRe currently supports custom code, SKLearn pipelines and the PyTorch framework within SKLearn pipelines.
- **Experiments** define the experimental setup consisting of a pipeline (i.e. the executable code), a dataset, and hyperparameters controlling the experiment as well as parameters controlling the experimental setup (e.g. splitting strategy)
- **Executions** are running of the workflows which are associated with a particular version of the source code. If the source code is changed, it results in a new execution.
- **Runs** are single executions of the pipelines. Each time the experiment is executed a new run is generated in the corresponding execution directory.
- **Computations** are the actual executions of a run, i.e. the execution of the workflow, over a dataset split.

In general, users do not have to care about Executions, Runs and Components. They need to implement their pipeline or machine learning component and wrap it with the wrapper PaDRe provides. Additionally, a experiment configuration needs to be provided including a dataset. When executing the experiment, PyPaDRE stores results and intermediate steps locally and adds it to the database of experiments. Afterwards, it provides easy means to evaluate the experiments and compare them, as outlined below.

For more details please refer to Setting up Experiments :ref:<setup_experiments>.

1.3.2 Components and Hyperparameters

Hyperparameters are distinguished between

- model parameters: parameters, that influence the model
- optimizer parameters: parameters, that influence the optimizer
- other parameters: parameters, not fitting into the above classes

Hyperparameters can be specified by the individual components directly in code (recommended for smaller experiments) or via a mappings file, which is a *json* file that links metadata to the implementation in a library. The mapping file also provides an extensible mechanism to add new frameworks easily. Via an inspector pattern padre can extract from relevant parameters and components from an instantiated pipeline.

Components follow some implementation details and provide *fit*, *infer* and configuration commands.

1.4 Experiment Evaluation

Experiments should store the following results

- **Raw Results** currently consisting of regression targets, classification scores (thresholded), classification, probabilities, transformations (e.g. embeddings).Results are stored per instance (per split).
- **Aggregated Results** are calculated from raw results. This includes precision, recall, f1 etc.
- **User Defined Metrics** are computed based on user provided code. The user can implement their own functions and wrap it with the PaDRe structure to provide custom metrics. This code is also versioned and stored as a code object.

Evaluation should include standard measures and statistics, but also instance based analysis.

1.5 Research Assets Management

Beyond experiment support, the platform should also help to manage research assets, like papers, software, projects research questions etc. Currently, these artifacts can be managed via adding them to the source code folder and let it be Git managed.

1.6 Metasearch and Automated Machine Learning

Not Yet Implemented

1.6.1 Python Class Interface

First, when knowing the details of all packages PyPaDRE can be used in code. This is either done by creating an `padre.experiment.Experiment` or through using decorators (currently under development). However, in this case the user is responsible for using the correct backends to persist results to.

```
app = example_app()

@app.dataset(name="iris",
         columns=['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
                  'petal width (cm)', 'class'], target_features='class')


def dataset():
data = load_iris().data
target = load_iris().target.reshape(-1, 1)
return np.append(data, target, axis=1)


@app.experiment(dataset=dataset, reference_git=__file__,
            experiment_name="Iris SVC - static seed", seed=1, project_name="Examples")
def experiment():
    from sklearn.pipeline import Pipeline
    from sklearn.svm import SVC
    estimators = [('SVC', SVC(probability=True))]
    return Pipeline(estimators)
```

Please note, that this is not the standard case and proper evaluation classes are currently under development.

1.6.2 Python App Interface

As a second interface, PyPaDRE support a high-level app. This high-level app integrates experiments, configuration files in a high level, easy to use interface.

```
from pypadre.core.model.project import Project
from pypadre.core.model.experiment import Experiment
from pypadre.binding.metrics import sklearn_metrics

self.app.datasets.load_defaults()
project = Project(name='Test Project 2',
                  description='Testing the functionalities of project backend',
                  creator=Function(fn=self.test_full_stack, transient=True,
                                   identifier=PipIdentifier(pip_package=_name.__name__
↪,
                                                            version=_version.__
↪version__)))

def create_test_pipeline():
    from sklearn.pipeline import Pipeline
    from sklearn.svm import SVC
    # estimators = [('reduce_dim', PCA()), ('clf', SVC())]
    estimators = [('SVC', SVC(probability=True))]
    return Pipeline(estimators)

id = '_iris_dataset'
dataset = self.app.datasets.list({'name': id})

experiment = Experiment(name='Test Experiment', description='Test Experiment',
                        dataset=dataset.pop(), project=project,
                        pipeline=SKLearnPipeline(pipeline_fn=create_test_pipeline,␣
↪reference=self.test_reference))
```

(continues on next page)

(continued from previous page)

```
experiment.execute(parameters={'SKLearnEvaluator': {'write_results': True}})
```

1.6.3 Python CLI Interface

The third interface is a command line interface for using Python via a command line. Please note that not all functions are available. Project and Experiments can be created via the CLI while computations, executions and runs can only be listed or searched. This is because the execution, runs, and computations have specific semantic meanings and are created while executing an experiment.

CHAPTER TWO

DESIGN PRINCIPLES

2.1 Identity Management

Every padre object that is stored/retrieved (e.g. dataset, experiment) will get a unique id. The id will be assigned by the backend. If an id is *None*, it has not been persisted by the backend.

2.2 Projects

A project is simply a collection of experiments. Each project has a unique name associated with it. The project name is the name of the directory in the file structure.

2.3 Experiment Naming and Uniqueness of Experiments

Every experiment is associated with a project. The experiment name has to be unique within a project. The experiment directory is created within each project directory, and the name of the directory is the experiment name

CHAPTER THREE

CONFIG

3.1 Padre Config

PadreConfig class covering functionality for viewing or updating default configurations for PadreApp. Configuration file is placed at ~/.padre.cfg

When running the tests ~/.padre-test.cfg is created which is deleted when the tests are completed.

Expected values in config are following

[LOCAL BACKEND]

root_dir = ~/.pypadre/

[GENERAL]

offline = True

oml_key = openml_key_here

3.2 Implemented functionality

1. Get all sections from config
2. Save config

3.3 Using config through CLI

1. Use **get [–param] [–section]** command to get value of given param
   - param: name of attribute
   - section: name of the section, if its None then default section will be used
2. Use **set [–param] [–section]** command to set value of given param
   - param: must be a tuple of key value pair
   - section: name of the section, if its None then default section will be used
3. Use **list [–section]** command to get list of all params for given section
   - section: name of the section

CHAPTER

FOUR

GETTING STARTED - FIRST STEPS WITH PYPADRE

4.1 Installation

Python Requirements - Python 3.7 Installation of the PyPaDRe client is then done by simply installing the pip package pip install pypadre

4.2 Configuration

When PaDRe is executed, a default configuration file is created in your home directory and in the current scenario the user does not have to add anything else. This config file can then be updated by the user with different configurations such as a GitHub token for logging the experiments to GitHub, a Database username and password for storing the experiments in a Database, or even the locations of datasets. The base config would usually contain whether PaDRe is offline and the root directory of PaDRe.

CHAPTER

FIVE

SETTING UP

5.1 Reference Object

Git is an important aspect of PaDRe because we need to version the source code for tracking the lifetime of experiments. Sometimes, the code might come from another python package too. For us to keep track of the provenance, we need a reference that points us to the source and this is done via the reference object. A reference object can either be a Python package or a Python file. If a Python file is given as a reference it should be part of a git repository. The source code git should be created by the user and this could be in any directory of the user's system environment. And if it is a Python package, we obtain the information about the Python package by inspecting the package and storing the package identifier and the function that is called for execution. The reference object is explicitly specified only when using pure code to create experiments and projects, while when using decorators it is automatically picked up by the PaDRe framework to resolve the required names and file objects into references.

5.2 Creating a Project in PaDRe

A project contains one or more experiments that are semantically grouped together. It could be either different experimental methods working towards an identical goal, or many different experiments that are parts of a larger goal.

Parameters of a project are - Name: Name of the project. - Description: A short description that provides information about the project. - Reference: A reference object that specifies how the project was created and how it is handled.

5.3 Creating an experiment in PaDRe

An experiment requires the following parameters to be initialized.

-Name: The name of the experiment. This should be unique -Description: A short description of the intention of the experiment. -Dataset: The dataset on which the experiment will work on -Pipeline: A workflow consisting of one or more algorithms -Project: The project to which this experiment belongs to. The name of the project can be specified and PaDRe automatically searches and groups the experiment under the specified project. -strategy: Splitting strategy for the dataset. The supported strategies are random, cv for cross validation, explicit where the user can explicitly specify the indices for training, testing and validation, function where the user passes a function that returns the indices or index where a list of indices are passed. If no option is given, the random splitting method is chosen. -preprocessing_pipeline: Preprocessing workflow for the dataset in a case that an algorithm has to be applied to the dataset as a whole for the experiment. This could be something such as computing the mean and standard deviation of a dataset or creating an embedding which normally should be based on the whole dataset. -reference: A reference object to the source code being executed

5.4 Single Pipeline Experiments

Single pipeline experiments can be created in different ways:

1. Through class instantiation

```
from pypadre.core.model.project import Project
from pypadre.core.model.experiment import Experiment
from pypadre.binding.metrics import sklearn_metrics
print(sklearn_metrics)

self.app.datasets.load_defaults()
project = Project(name='Sample Project',
                  description=Example Project',
                  creator=Function(fn=self.test_full_stack))

_id = '_iris_dataset'
dataset = self.app.datasets.list({'name': _id})

experiment = Experiment(name='Sample Experiment', description='Example Experiment',
                        dataset=dataset.pop(), project=project,
                        pipeline=SKLearnPipeline(pipeline_fn=create_test_pipeline_
↪multiple_estimators),
                        reference=self.test_full_stack)
experiment.execute()
```

2. Through decorators

```
import numpy as np
from sklearn.datasets import load_iris

# Import the metrics to register them
from pypadre.binding.metrics import sklearn_metrics
from pypadre.examples.base_example import example_app

app = example_app()


@app.dataset(name="iris",
             columns=['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)',
                      'petal width (cm)', 'class'], target_features='class')
def dataset():
    data = load_iris().data
    target = load_iris().target.reshape(-1, 1)
    return np.append(data, target, axis=1)


@app.experiment(dataset=dataset, reference_git=__file__,
                experiment_name="Iris SVC - User Defined Metrics", seed=1, allow_
↪metrics=True, project_name="Examples")
def experiment():
    from sklearn.pipeline import Pipeline
    from sklearn.svm import SVC
    estimators = [('SVC', SVC(probability=True))]
    return Pipeline(estimators)
```

3. Creating an experiment via the CLI

pypadre > project create –name PROJECT_NAME pypadre > experiment initialize –name EXPERIMENT_NAME The above command opens an editor where the user eidt the code for the dataset and experiments similar to that of the decorator example

pypadre > experiment execute –name EXPERIMENT_NAME (alternatively, the user can use the path to the experiment with –path)

5.5 Hyperparameter Optimization

1. Through parameters passed to the experiment execute function. The parameters are passed as a dictionary with the key as the component name and an inner dictionary. The inner dictionary contains the parameter name as the key and an array of values that are to be used for hyperparameter optimization.

```
parameter_dict = {'SVR': {'C': [0.1, 0.2]}}
experiment.execute(parameters={'SKLearnEvaluator': {'write_results': True},
↪'SKLearnEstimator': {'parameters': parameter_dict}
```

2. Through decorators using the parameter keyword

```
@app.dataset(name="iris", columns=['sepal length (cm)', 'sepal width (cm)', 'petal␣
↪length (cm)',
                                   'petal width (cm)', 'class'], target_features=
↪'class')
def dataset():
    data = load_iris().data
    target = load_iris().target.reshape(-1, 1)
    return np.append(data, target, axis=1)

@app.parameter_map()

def parameters():
    return {'SKLearnEstimator': {'parameters': {'SVC': {'C': [0.1, 0.5, 1.0]}, 'PCA':
↪{'n_components': [1, 2, 3]}}}}

@app.experiment(dataset=dataset, reference_package=__file__, parameters=parameters,␣
↪experiment_name="Iris SVC",
                project_name="Examples", ptype=SKLearnPipeline)
def experiment():
    from sklearn.pipeline import Pipeline
    from sklearn.svm import SVC
    estimators = [('PCA', PCA()), ('SVC', SVC(probability=True))]
    return Pipeline(estimators)
```

5.6 Multi-pipline, multi-data Experiments

Currently, not supported

CHAPTER

SIX

LOGGING IN PADRE

Padre has the ability to log to multiple different locations such as file, http, database etc at the same time provided there is a backend define for it. Currently, the logging is only done to the file backend, which is then taken up by the Git backend when the experiment is committed. The file backend writes incoming messages to the local disk at a central location. The user can easily extend the logging functionality to incorporate their own code for user specific handling of events and logging. A basic framework is provided and the user can add new backends and handle each event based on their custom backends. This is useful when the user has experiments running on a cluster and needs to write the experiments to a slack service or use python code to periodically notify the user about the experiment updates.

The incoming logging messages can be of three levels of logging - INFO: These are the normal messages which are logged, like the start of an experiment, the beginning of different stages and its completion along with the respective time stamps

- WARN: These are warning information which could lead to the framework crashing if the user does not heed to the

messages. For example if the dataset has no target but is loaded, a warning message appears saying that this particular dataset should not be used for supervised learning.

- ERROR: These log messages are of critical importance and will disrupt the proper functioning of the framework and

cause it to crash. A condition is passed along with the log error event and if the condition is satisfied all the backends are written with the error information and the Padre framework stops its execution.

Currently, the logging is done via firing of events via the blinker package in Python. The fired event is then handled at the service which then passes it to the backend. The backend then logs the event as per defined. For the FileBackend the log format of the message is current time + LOG_LEVEL(INFO, WARN or ERROR) followed by the message.

CHAPTER

SEVEN

METRICS

Metrics are an important part of an experiment. It allows us to compare different experiments and make informed decisions. In PaDRe, metrics are a very important part of running experiments. The users can compute different metrics and basic metrics are provided PaDRe itself, such as confusion matrix, precision, recall, f1 measure and accuracy for classification and mean error, mean absolute error etc for regression.

Metrics can be computed at the output of any component. For example, the user might want to know the average distance between points in an embedding space and this should be computed after the preprocessing step. PaDRe supports computation of intermediate metrics from components. All the metrics are stored in a metrics registry as metric objects. Each metric object has a field that specifies what type of data it would consume. For example, for precision it would be a confusion matrix. A matrix graph is constructed by chaining the producer of metrics and the next consumer. PaDRe supports branching of metrics too.

During execution of each component, PaDRe checks whether there are any available metrics for that particular component and if there are available metrics, PaDRe prints out the available metrics to the user.

If the user wants to add a custom metric, all the user has to do is wrap the function for computing the metric in a metric object, specify what the metric object would consume and add the metric object to the metric registry. The user also has to create a reference to code for logging the version of the code.

And if the user wants only a specific metric that can also be accomplished in PaDRe. For example, if the user needs only the confusion matrix, the user specifies that only this metric is needed and PaDRe computes all the simple paths in the graph for that component that leads up to that metric and executes only those paths.

```
ACCURACY = "Accuracy"


# Defining the function how the metric is computed
def accuracy_computation(ctx, option='macro', **kwargs):
    (confusion_matrix_metric, computation) = unpack(ctx, "data", "computation")
    confusion_matrix = confusion_matrix_metric.result

    """
    This function calculates the accuracy
    :param confusion_matrix: The confusion matrix of the classification
    :param option: Micro averaged or macro averaged

    :return: accuracy
    """
    if confusion_matrix is None:
        return None

    metrics = dict()
    precision = np.zeros(shape=(len(confusion_matrix)))
    recall = np.zeros(shape=(len(confusion_matrix)))
```

(continues on next page)

(continued from previous page)

```
    f1_measure = np.zeros(shape=(len(confusion_matrix)))
    tp = 0
    column_sum = np.sum(confusion_matrix, axis=0)
    row_sum = np.sum(confusion_matrix, axis=1)
    for idx in range(len(confusion_matrix)):
        tp = tp + confusion_matrix[idx][idx]
        # Removes the 0/0 error
        precision[idx] = np.divide(confusion_matrix[idx][idx], column_sum[idx] +␣
↪int(column_sum[idx] == 0))
        recall[idx] = np.divide(confusion_matrix[idx][idx], row_sum[idx] + int(row_
↪sum[idx] == 0))
        if recall[idx] == 0 or precision[idx] == 0:
            f1_measure[idx] = 0
        else:
            f1_measure[idx] = 2 / (1.0 / recall[idx] + 1.0 / precision[idx])

    accuracy = tp / np.sum(confusion_matrix)
    if option == 'macro':
        metrics[ACCURACY] = accuracy

    elif option == 'micro':
        metrics[ACCURACY] = accuracy

    else:
        metrics[ACCURACY] = accuracy

    return Metric(name=ACCURACY, computation=computation, result=deepcopy(metrics))


# Defining the class that contains the metric, its name and what it consumes
class Accuracy(MetricProviderMixin):

    NAME = "Accuracy"

    def __init__(self, *args, **kwargs):
        super().__init__(code=PythonPackage(package=__name__, variable="accuracy_
↪computation",
                                            repository_identifier=PipIdentifier(pip_
↪package=_name.__name__,
                                                                                   ␣
↪version=_version.__version__)),
                                                                                   ␣
↪**kwargs)

    @property
    def consumes(self) -> str:
        return str(ConfusionMatrix)

    def __str__(self):
        return self.NAME


# Defining the reference to the metric
accuracy_ref = PythonPackage(package=__name__, variable="accuracy",
                             repository_identifier=PipIdentifier(pip_package=_name.__
↪name__,
                                                                   version=_version.__
↪version__))
```

(continues on next page)

(continued from previous page)

```

# Creating the metric object
accuracy = Accuracy(reference=accuracy_ref)

# Adding it to the metric registry
metric_registry.add_providers(accuracy)
```

In addition to all this, there is a flag that is known as allow_metrics. This flag allows the user to turn off the metric computation if there is ever such a scenario. The allow_metric flag is set as true by default but can be turned off on a component level basis.

CHAPTER

EIGHT

VISUALISATIONS

8.1 Dataset visualization

8.1.1 Scatter Plot

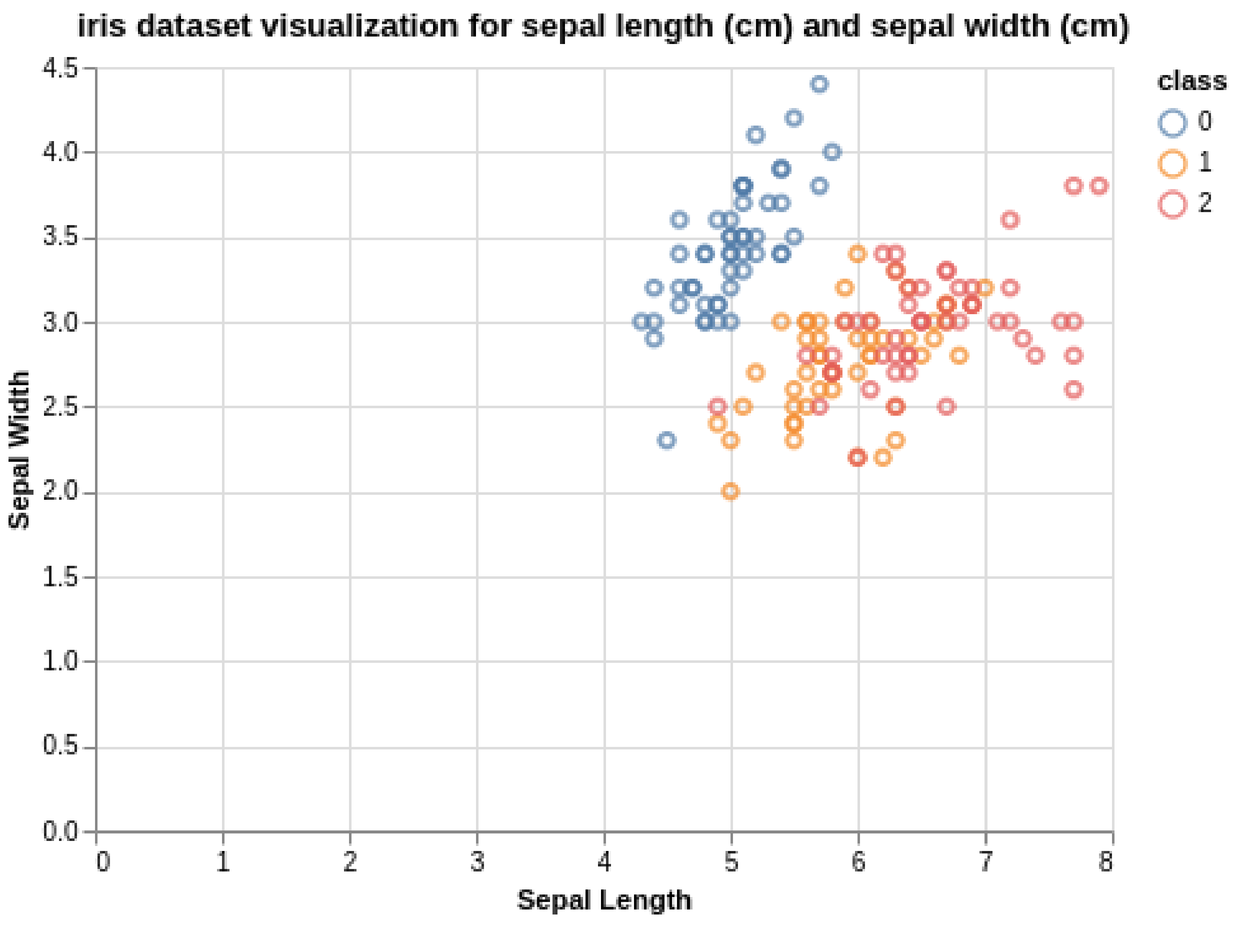
PyPaDRe supports different visualizations for the dataset as well as results. For example, Scatter plot for the attributes of dataset can be created in the form of json specification(vega lite). For creating visualization altair library is used. To create scatter plot `pypadre.core.plot.Plot.get_scatter_plot()` function is provided in the plot module which should be called with dataset attributes of the users choice.

```
ds = dataset
app.backends[0].dataset.put(ds, allow_overwrite=True)  # Save dataset locally


plt = plot.DataPlot(ds)
vis = plt.get_scatter_plot("sepal length (cm)",
                           "sepal width (cm)",
                           "Sepal Length",
                           "Sepal Width")

# Save visualization locally for above dataset
app.backends[0].dataset.put_visualization(vis,
                                          file_name="scatter_plot.json",
                                          base_path=os.path.join(app.backends[0].
↪dataset.root_dir, ds.name))
```

Note: :*pypadre.core.plot.Plot.get_scatter_plot* takes x attribute and y attribute as required arguments for which plot should be created.


iris dataset visualization for sepal length (cm) and sepal width (cm)
class
0
1
2
Sepal Width
Sepal Length
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
1
2
3
4
5
6
7
8

CHAPTER

NINE

PYPADRE DEVELOPMENT

9.1 Module Architecture

9.1.1 PyPadre App

PyPaDRe provides a CLI to interact with PyPaDRe and this is done via apps. There are different apps within PyPaDRe such as the Project App, Experiment App and so on. Apps provide a method to interact with components of PyPaDRe. The apps support different functions such as listing, searching, and deleting. There are different apps such as - project app - experiment app - execution app - run app - computation app - dataset app - metric app - computation app - config app

The PaDRe app is the main app and interfaces with all the other apps. These apps provide functionalities regarding their modules to the user.

9.1.2 PyPadre CLI

The PyPadre CLI is the command line interface to the PyPadre App. The app can be invoked from the command line by typing "pypadre". Projects, experiments, executions, runs, and computations can be accessed from the CLI.

- list: Supported by all the modules. It simply lists all the modules at which it is called. For example: if the list

is used in conjunction with experiment("experiment list"), all the experiments are listed.

- get: Supported by all the modules. It loads the object into the memory.
- active: Supported by all modules except computation. This commands sets an active module. For example, setting an

active project so that all the following commands can take the active project as the one set by the user.

- create: Supported by project alone. This command is used to create new projects.
- initialize: Supported by the experiment module alone. This is done to initialize the experiment with the required

parameters.

- execute: Supported by the experiment module alone. This command is used to execute an experiment once it has been

initialized.

- load: loads a dataset from the given source
- sync: this command is used to synchronize the datasets among the different backends
- compare: Supported by the experiment and execution modules only. For experiments, this is used to compare the metadata

and the pipelines of two experiments. For executions, this is used to compare the source code which is referencing each execution.

- compare_metrics: supported by the metric app to compare the metrics of two different runs
- get_available_estimators: lists all the available estimators for the specified experiment
- list_experiments: lists all possible experiments that can be compared
- reevaluate_metrics: reevaluates all the metrics from the results of the experiment
- set: lets the user set a key value pair in the current configuration

Unit Testing and Examples

- Unit tests for each module are placed within those modules.
- Tests must use the prefix *tests_*
- Experiments created during tests will be removed when the tests are completed. It is the same for the configuration files too.
- Unit tests should be written as often as possible (i know, its a pain) and as a proof of concept.

CHAPTER

TEN

EXPERIMENT STORAGE AND ONLINE SYNCHRONIZATION

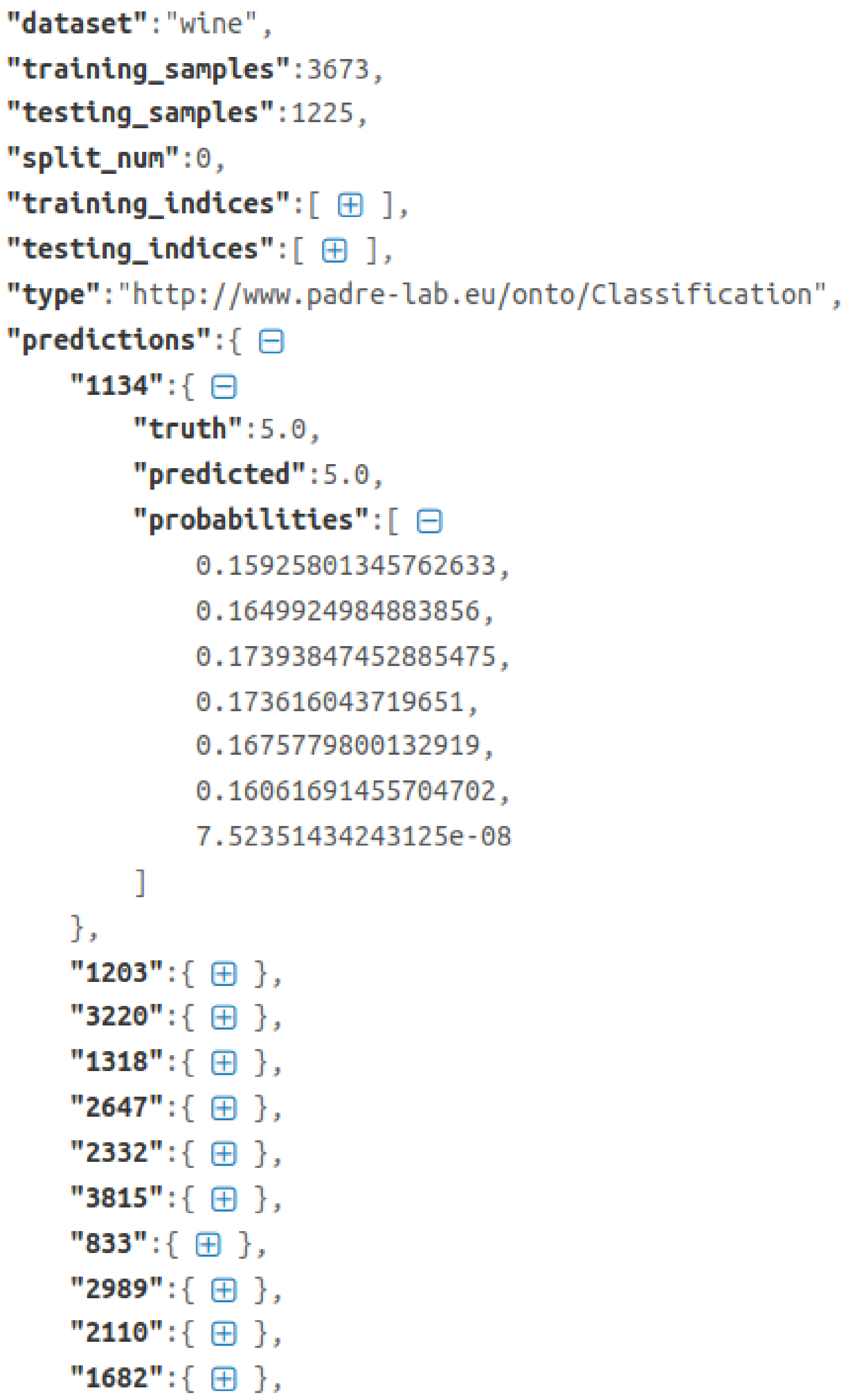
10.1 Local Storage Structure

Git is the backbone of PaDRe. Every experiment, project and dataset is a git repository. Datasets are added with Git LFS. Each experiment is a submodule of the project. For every experiment, a source code version is associated with the experiment. Every time the source code changes, a new execution is created. Every execution of the experiment is a run. A run is defined as the execution of the pipeline with a dataset. Runs are contained in executions so that every particular run is associated with a specific version of the source code. Within each run, there are computations which are nodes in the pipeline. The metadata and if needed output of the nodes are stored within these computations.

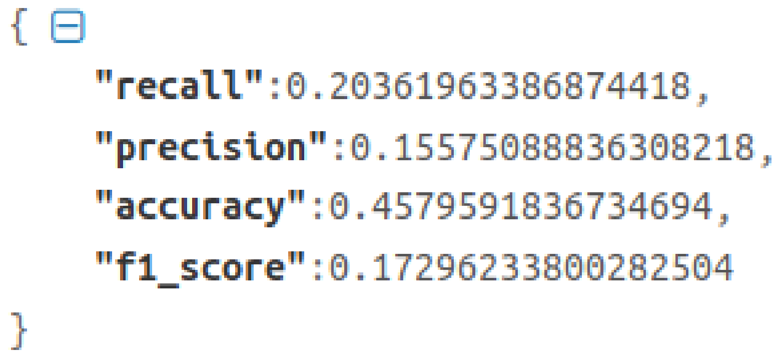
10.2 Storing of results

The storing of results is mostly defined from the viewpoint of the FileBackend. The Git backend uses the stored results from the FileBackend to add the files to the repository. The HTTPBackend is not discussed.

At the end of training the model, and if there are test indices, PaDRe tests the model using the dataset rows that are specified using the test indices. The results are then stored in a JSON file containing the dataset name, the length of the training and testing indices, the split number(for cross validation), and for each row in the testing dataset, PaDRe stores the truth value, the predicted value and the probabilities. This is done because the user can easily recompute the results if needed, or extend the experiment using custom metric evaluations.



```
"dataset":"wine",
"training_samples":3673,
"testing_samples":1225,
"split_num":0,
"training_indices":[ ⊞ ],
"testing_indices":[ ⊞ ],
"type":"http://www.padre-lab.eu/onto/Classification",
"predictions":{ ⊟
    "1134":{ ⊟
        "truth":5.0,
        "predicted":5.0,
        "probabilities":[ ⊟
            0.15925801345762633,
            0.1649924984883856,
            0.17393847452885475,
            0.173616043719651,
            0.1675779800132919,
            0.16061691455704702,
            7.52351434243125e-08
        ]
    },
    "1203":{ ⊞ },
    "3220":{ ⊞ },
    "1318":{ ⊞ },
    "2647":{ ⊞ },
    "2332":{ ⊞ },
    "3815":{ ⊞ },
    "833":{ ⊞ },
    "2989":{ ⊞ },
    "2110":{ ⊞ },
    "1682":{ ⊞ },
```



```
{
    "recall":0.20361963386874418,
    "precision":0.15575088836308218,
    "accuracy":0.4579591836734694,
    "f1_score":0.17296233800282504
}
```

At the end of training the model, and if there are test indices, PaDRe tests the model using the dataset rows that are specified using the test indices. The results are then stored in a JSON file containing the dataset name, the length of the training and testing indices, the split number(for cross validation), and for each row in the testing dataset, PaDRe stores the truth value, the predicted value and

10.3 Server Synchronisation

The server supports uploading and downloading of experiments but was developed as a Proof Of Concept. It is not addressed in this documentation.

10.3.1 Versioning

Versioning of PaDRe is done via Git. The user can use the Git to track the lifecycle of the experiment.

CHAPTER

ELEVEN

BACKEND

11.1 Backend

Backend is used to communicate experiments, runs and splits, metrics and results information. Following are some important functions implemented in this backend. Backends handle flow of information from the pypadre framework to the outside world. Backends are used for Creating, Updating, Reading, and Deleting the different PaDRe objects. Backends can be a File Backend, HTTP Backend, Git Backend etc

11.1.1 Http Backend

Http backend is used to exchange information between client and server. Base class for Http backend is used
to setup server information, authenticate user, check if client is online, retrieve token and do generic http calls. It was implemented as a Proof Of Concept and is not included currently in the package.

11.2 Backend Functionalities

Backends link to each part of the experiment such as projects, experiments, executions, runs and computations. All the backends provide get functionalities to retrieve the individual modules. Backends hold the repositories within them. The repositories contain the logic to persist the data whether it is on File, Database, HTTP etc.

For example the functionality for experiments described below:

1. put_experiment: It receives an experiment. If it is an instance of the experiment class, the experiment class is first stored to disk using the file backend. If it is a string with the name, the experiment is loaded from the file backend and stored to the serer. The experiment in the file backend should receive an url in its metadata file to indicate that it has been uploaded to the server.

2. delete_experiment: Deletes the experiment given by ex (either as class or as string) from the local file cache or the remote server. It also receives mode whose possible values can be all, remote or local

3. get_experiment: Downloads the experiment given by ex, where ex is a string with the id or url of the experiment. The experiment is downloaded from the server and stored in the local file store if the server version is newer than the local version or no local version exists. The function returns an experiment class, which is loaded from file.

11.3 Communication to the Backend

Communication with the backends are done via the service layer. The service of each module such as the Dataset, Experiment, Project, Execution call their corresponding services via events. The services then access the backends and execute the functions for listing objects, loading objects, etc.

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search