pailab Documentation

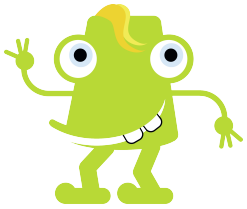
Release 0.4.3

pailabteam

Oct 10, 2019

Contents:

CHAPTER 1



pailab

pailab is an integrated machine learning workbench to version, analyze and automatize the machine learning model building processes and deployments. It keeps track of changes in your machine learning pipeline (code, data, parameters) similar to classical version control systems considering special characteristics of AI model building processes.

It provides:

- Versioning of all objects of the ML development cycle
- Full transparency over the whole ML development cycle
- Support to work in a team on the same AI projects, sharing data, algorithms and results
- Consistency checks across the whole pipeline
- Distributed execution of parallel jobs, e.g. for parameter studies
- Standardized analysis plots
- Jupyter widgets to administrate and control the ML repo

Al objects added to the repository are split into a part containing big data and a part with the remaining data handling them separately with different technqiues. For example one may use git to administrate the remaining data part. It also adds information to each object such as the version, the author, date etc. and each object is labeled with a category defining the role in the ml process.

1.1 Overview

1.1.1 Install

Install pailab using pip:

```
pip install pailab
```

1.1.2 Source code

pailab's source code is available on GitHub:

https://github.com/pailabteam/pailab

and cloned using:

```
git clone https://github.com/pailabteam/pailab.git pailab
```

1.1.3 Examples and first steps

Tutorial

To learn how to work with pailab you may find the *Tutorial* useful. The tutorial gives an introduction to all building blocks and tools.

Notebooks

If you like working with jupyter, there are the following jupyter notebooks demonstrating pailab's functionality. The notebooks are located in the examples folder of pailab's GitHub repo. Please note that the plots in these notebooks are created using plotly. Therefore if you want to play around with the plotting functionality you have to install this. However, even if you do not want to install plotly, the notebooks are nevertheless a very good starting point.

Introductionary

- boston_housing.ipynb: Shows pailab's basic functionality using the boston housing data set and a regression tree from scikit learn (without preprocessing).
- adult-census-income.ipynb: Shows pailab's basic functionality using the adult-census data set and a regression tree from scikit learn (including preprocessing).
- boston_housing_widgets.ipynb: Similar to boston_housing.ipynb but using some jupyter widgets.
- boston_housing_distributed.ipynb: Similar to boston_housing.ipynb but using the `pailab.job_runner.SQLiteJobRunner` job runner to execute jobs in a different thread or even on a different machine.

Advanced

- caching_demo.ipynb: Simple example demonstrating the caching of results of time consuming functions.
- Convolutional_Autoencoder.ipynb: Notebook from Udacity's Deep Learning Nanodegree program GitHub repository modified to use pailab. Here, an autoencoder for the MNIST dataset is built using PyTorch.
- Convolutional_Autoencoder_disk_store.ipynb: Same as Convolutional_Autoencoder.ipynb but using disk store and hdf5 files to persist all objects created during the analysis of the problem

1.1.4 Logging

pailab uses the Python standard logging module `logging` making use of different log levels. The log level of the logger may be globally set by:

```
import logging as logging
logging.basicConfig(level=logging.DEBUG)
```

See the `logging` modules documentation for further details.

1.1.5 How to File a Bug Report

Bug reports are always welcome! The issue tracker is at:

https://github.com/pailabteam/pailab/issues

Please always include pailab's version into the issue:

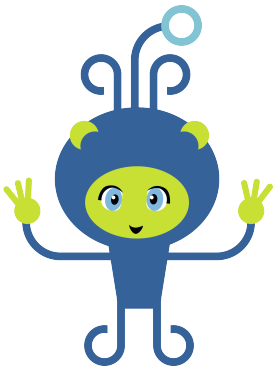
```
import pailab
print(pailab.version.info)
```

1.1.6 Laby and Bugy

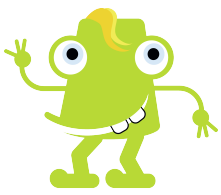
During a trans-universal trip in 2500 Laby and Bugy made a stop on earth. They were quite astonished to see how far humans had developed the AI business but they got a little frightened when they saw how blind-folded humans worked in this business. At least Laby would have not been far from a heart attack if he would have had a thing we humans might call heart. They soon decided to help these poor underdeveloped human species and to make a little time travel to the beginning of the AI bubble. So, when they arrived in January 2019, they started to develop pailab.

They form quite a good team, complementing each other. Laby fights each bug with her weapons brought from her home planet Labmania, documenting everything (Bugy had to put a lot of effort convincing her not to document the documentation) and producing code passing all beautifiers for all styleguids without being changed (and has overruled Chuck Norris since his code was not matlab compliant). She is really enthusiastic about testing.

Bugy is more the chaotic but creative alien. He loves to produce many new functionalities and hates documenting. He has not understand the sense of measuring test coverage but he may implement a better Powerpoint application then the one implemented by his cousin Bugsy (who uses the initials MS for whatever reason) with maybe three lines of code (if we count his comment to understand his program the source code file might have four lines).



Laby and



Bugy

1.2 Basics

In this section we explain pailab's basic building blocks. For a quick introduction how to work with pailab and to get a first impression of the functionality we refer to work to the *Repo initialization, training, evaluation*.

1.2.1 Overview

pailab's core is the *pailab.ml_repo.repo.MLRepo* class which is what we call the machine learning repository. The repository stores and versions all objects needed in the machine learning development cycle. There are three fundamental differences to version control systems such as git or svn for classical software development:

- Instead of source code, objects are checked into the repository. Here, each object must inherit or at least implement the respective methods from *pailab.ml_repo.repo_objects.RepoObject* so that it can be handled by the repository. Furthermore, each such object belongs to a certain category (*pailab.ml_repo.repo.MLObjectType*), so that the repository may perform certain checks and allow to automatize the ml build pipeline.
- Each object is split into a part with standard data and a part with (large) numerical data and both parts are stored separately in different storages. Here, the normal data is stored in a storage derived from *pailab.ml_repo.repo_store.RepoStore* whereas the numerical data is stored via a *pailab.ml_repo.repo_store.NumpyStore*.
- The execution of different jobs such as model training and evaluation or error computation is triggered via the MLRepo. Here, the MLRepo simply uses a JobRunner to execute the jobs.

As we see, we need three ingredients to initialize an MLRepo:

- RepoStore (handles the object data)
- NumpyStore (handles numpy part of the object data)
- JobHandler (runs the jobs such as training or evaluation)

1.2.2 Basic functionality

The MLRepo offers four main functionalities

- Adding objects *pailab.ml_repo.repo.MLRepo.add()*
- Retrieving objects *pailab.ml_repo.repo.MLRepo.get()*
- Running Jobs (which are also objects from the repo perspective and therefore stored in the repo) *pailab.ml_repo.repo.MLRepo.run()*
- Listing objects by their category *pailab.ml_repo.repo.MLRepo.get_names()*

All other methods are just using these methods and provide a little bit more convenience in daily work.

Add objects

When adding an object to the repository the MLRepo automatically adds additional information to the object, stored in the attribute repo_info. The repo_info attribute is an instance of the *pailab.ml_repo.repo_objects.RepoInfo()* and contains information such as the version number (this number is autogenerated from MLRepo), the objects name, a description of the object, a commit message, the author i.e. the user who added the object. So to add an object to an instance ml_repo of the MLRepo class you just call:

```
ml_repo.add(obj, message = "just an example")
```

which adds the object where `message` is attached to the object in the `repo_info` attribute as well. Note that the method returns the version which was attached to the object so that:

```
version = ml_repo.add(obj, message = "just an example")
```

`version` contains the version number after the execution.

We can also add multiple objects at the same time storing them in a list, that is:

```
versions = ml_repo.add([obj1, obj2], message = "just an example")
```

where `version` is now a list of versions for the different objects added.

Get objects

To retrieve objects from the storage one can use the *pailab.ml_repo.repo.MLRepo.get()* method. Here, different possibilities to specify what object to get exist. If one wants to retrieve a special version of an object you can call get with the object's name and the specified version:

```
obj = ml_repo.get('obj_name', version = 'aasfdg-111-ezrhf')
```

If one wants to retrieve the first or last version, instead of typing the specific cryptic version string, we may use the keywords `repo_store.repo.RepoStore.LAST_VERSION` and `repo_store.repo.RepoStore.FIRST_VERSION`:

```
obj = ml_repo.get('obj_name', version = RepoStore.FIRST_VERSION)
```

returns the first version of the object with name `obj_name`. Instead of specifying a single version, we can also retrieve a bunch of different versions just using a list of versions:

```
objs = ml_repo.get('obj_name', version = ['aasfdg-111-ezrhf', RepoStore.FIRST_
↪VERSION])
```

returns a list of the different versions of the object with name `obj_name`.

Another frequent use case is to retrieve an object that has been created using a different object with a specific version. For example, you may be interested to get mean squared error (mse) for a specific model. Then, you can just use get to retrieve the mse object containing the mse for the specified model by calling get specifying the modifier_versions:

```
obj = ml_repo.get('obj_name', version=None,  modifier_versions={'obj2_name': 'aasfdg-
↪111-ezrhf'})
```

which returns the object with name obj_name and the version which has been constructed using the object 'obj2_name' with the specified version. Analogously to above one can also ask for all objects which haven been constructed using an object with object's version in a list of specified versions, i.e.:

```
objs = ml_repo.get('obj_name', version=None,  modifier_versions={'obj2_name': [
↪'aasfdg-111-ezrhf', 'bbhuuu-123-ooo'})
```

Important: If an object does not exist, the method may either throw an exception or return an empty list, depending on the argument **throw_error_not_exist**, i.e.:

```
obj = ml_repo.get('obj_name', version = 'aasfdg-111-ezrhf', throw_error_not_exist =␣
↪True)
```

throws an exception if an object with this name and version does not exist.

If we just want to check if exactly one object with a specified version or modification information exists, we may call the method setting the argument `throw_error_not_unique` to `True`, which means that an exception is thrown if there are more then one object satisfying the condition.

As we have discussed, an object is split into two parts that are stored separately: The 'small' data and the 'big' data of the object. By default, the get method returns the object leaving out the big data part. If one wants to get the complete object, one must set the argument `full_object` to `True`:

```
ml_repo.get('obj_name', version = 'aasfdg-111-ezrhf', full_object = True)
```

Getting names

To list al objects of a certain category stored in the repo, we can use *pailab.ml_repo.repo.MLRepo.get_names()*::, e.g. to get the ids of all test data objects in the repo

names = ml_repo.get_names(MLObjectType.TEST_DATA)

Running Jobs

All functions which may be called from the MLRepo, e.g. the function to train a certain model, are also objects which are stored within the repo. Typically, these objects inherit from the `pailab.ml_repo.repo.MLRepo.Job`. To run such a job using the repo, you can use `pailab.repo.MLRepo.run()` method:

```
job_info = ml_repo.run(job)
```

This line of code will add the Job object `job` to the repo and then call the `add` method of the ml_repo's internal `JobRunner`. The `job_info` variable will either be a tuple of the job's name and the job's version number in the repo or a string containing a message that no input data has been changed since last run of the job.

Note: Before the `run` method adds the job to the repo, it checks if the job has a method `check_rerun` and if so, calls it to decide if the job really has to be executed (if the method returns `True`). If the result of `check_rerun` is `False`, the method does not run the job.

Tip: Note that for many situations such as training or evaluating a model there are respective methods wrapping *pailab.ml_repo.repo.MLRepo.run()* which should be preferred. However, all these method will, after creation of the needed `Job` object, at the end call `run`.

Add a model

A model is defined by specifying

- preprocessing methods (in a certain order, optional)
- a function to evaluate the calibrated model on a given dataset
- a function to train the model given data, training and model parameter

- training parameter
- model parameter (optional)

This specification is done by setting the identifies of the objects into an instance of `pailab.repo_objects.Model`. Let us assume you have added a preprocessor with name `'uniform_scaling'`, a function to evaluate the model named `'eval'`, a function to train the model `'train'` and training parameter `'training_param'`, you can then add a new model to the repo by:

```
model = repo_objects.Model(preprocessors = 'uniform_scaling',
                           eval_function='eval', train_function='train',
                           training_param='training_param')
model.repo_info.name = 'name_of_model'
ml_repo.add(model, message='my first model')
```

As we see in this example, you do not have to specify a model parameter if your training function does not need one. Also, you do not have to specify the preprocessing, if you do no want to apply a preprocessing technique. Another possibility is to use the method `add_model` that may be more convenient:

```
ml_repo.add_model('name_of_model', model_eval = 'eval', model_training = 'train',
                  training_param = 'training_param',
                  preprocessors = 'uniform_scaling')
```

If there is only one evaluation function in the MLRepo and you want to use it, the method `add_model` will do the job for you, i.e. just do not specify the evaluation function:

```
ml_repo.add_model('name_of_model', model_training = 'train',
                  training_param = 'training_param',
                  preprocessors = 'uniform_scaling')
```

which now checks if there is only one eval function in the ml_repo and in this case uses this unique function. If there is more then one function or no function, an exception is thrown. This logic applies to all members except the preprocessing: If you do not define a preprocessing, no preprocessing will be applied. So, to specify a model under the assumption that the respective components are in the repository using the preprocessing from above, we may call:

```
ml_repo.add_model('name_of_model', preprocessors = 'uniform_scaling')
```

1.2.3 Setting up an MLRepo

In-memory

The easiest way to start using pailab is instantiate MLRepo using all defaults, except the user which must be specified, otherwise an exception is thrown.

```
        ml_repo = MLRepo(user = 'test_user')
```

This results in an MLRepo that handles everything in memory only, using *pailab.ml_repo.memory_handler.RepoObjectMemoryStorage* and *pailab.ml_repo.memory_handler.NumpyMemoryStorage* so that after closing the MLRepo, all data will be lost. Therefore this should be only considered for testing or rapid and dirty prototyping. Note that in this case, the JobRunner used is the *pailab.job_runner.job_runner.SimpleJobRunner* which simply runs all jobs sequential on the local machine in the same python thread the MLRepo has been constructed (synchronously).

Disk

To initialize an MLRepo so that the objects are stored on disk, we need to setup the respective storages within the MLRepo. One way to achieve this is to define the respective configurations in a dictionary and initialize the MLRepo with this dictionary. An example for such a configuration is given by

```
config = {
  'user': 'test_user',
  'workspace': 'tmp',
  'repo_store':
  {
      'type': 'disk_handler',
      'config': {
          'folder': 'tmp/objects',
          'file_format': 'json'
      }
  },
  'numpy_store':
  {
      'type': 'hdf_handler',
      'config':{
          'folder': 'tmp/repo_data',
          'version_files': True
      }
  },
  'job_runner':
  {
      'type': 'simple',
      'config':{}
  }
}
```

First we see that there is a user and also a workspace defined in the dictionary. The workspace is a directory where the configuration and settings are stored so that when you instantiate the MLRepo again, you just need to specify the workspace and not the whole settings again. The RepoStore used within the MLRepo is defined via the dictionary belonging to the repo_store key. Here we see that the configuration consists of describing the type of store (here we use the disk_handler which simply stores the objects on disk) and the settings for this storage. In our example the objects are stored in json format in the folder example_1/objects. The NumpyStore internally used is selected so that the big data will be stored in hdf5 files.

Now we simply instantiate the MLRepo using this configuration.

```
ml_repo = MLRepo(config = config)
```

To instantiate the MLRepo and directly save the respective config you have to set the parameter save_config

```
ml_repo = MLRepo(config = config, save_config=True)
```

Saving the config you may instantiate the MLRepo another time simply by

```
ml_repo = MLRepo(workspace = 'tmp')
```

git

The previous example stored the objects simply as json files on disk. There is the possibility to use git to manage the files. Here, you just have to replace the type by 'git_handler', i.e. just change in the configuration dictionary

above the type from `disk_handler` to `git_handler`. The MLRepo will then use the *pailab.ml_repo.git_handler.RepoObjectGitStorage* as repo.

If you have a remote git repository which you want to use, you have to clone the repository first and then specify the directory of the cloned repo as directory of the git_handler.

1.2.4 Integrate a new model

As we have seen in *Add a model*, a model needs

- preprocessing methods
- a function to evaluate the calibrated model on a given dataset
- a function to train the model given data, training and model parameter
- training parameter
- model parameter (optional)

When a model is fully specified, you can call `pailab.repo.MLRepo.run_training()` to train the model. The calibrated model as a result of the training function is then stored within the repo in the category `CALIBRATED_MODEL`. Therefore, to integrate a new model, you have to specify the two methods to train and evaluate the model, the training parameter class (if needed also the model parameter) and finally the object containing the calibrated model.

Example

In this section we show one way to get your custom model into pailab using a very simple and , we discuss a very simple, illustrative and Let us assume that you have implemented a new ml algorithm encapsulated in a class `SuperML`

```
class SuperML:
    @repo_object_init()
    def __init__(self):
        self._value = None

    def train(self, data_x, data_y, median = True):
        if median:
            self._value = np.median(data_y)
        else:
            self._value = np.mean(data_y)

    def eval(self, data):
        return self._value
```

This algorithm either uses the mean computed on a given dataset or the median depending on the given Boolean argument `median` as a forecast (a very simple algorithm :-)). Note that we have already used the `@repo_object_init()` decorator from `pailab.repo_objects` to add the methods and attributes needed to use this class within the repo. Another possibility would have been to directly implement the respective methods and attributes within the class on ourselves. See `pailab.repo_objects.RepoObject` which defines the respective interface. A third alternative would have been to implement a wrapper class which simply contains our `SuperML` class and implements all RepoObject functionality (an example for this can be found in the *pailab.externals.tensorflow_keras_interface* module). The way you choose depends on your flavor as well as on the question if the ML algorithm is your own development or if you may be wrapping just another ML module.

So, what is missing to put the new model into the MLRepo are the eval and train functions as well as the class storing the training parameter (which is simply one Boolean).

```
class SuperMLTrainingParam:
    @repo_object_init()
    def __init__(self):
        self.median = True
```

```
def train(training_param, data_x, data_y):
    result =  SuperML()
    result.train(data_x, data_y, training_param.median)
    return result
```

```
def eval(model, data):
    return model.eval(data)
```

After we defined the respective functions, we have to expose their definitions to the `MLRepo`. Here, we could either construct the respective objects `pailab.repo_objects.Function` using their special categories `MODEL_EVAL_FUNCTION` or `TRAINING_FUNCTION` or just use `pailab.repo.MLRepo.add_eval_function()` and `pailab.repo.MLRepo.add_training_function()`

```
            ml_repo.add_eval_function(train,
                                      repo_name='my_eval_func')
            ml_repo.add_training_function(eval, repo_name='my_eval_func')
```

In addition, we add a first set of training parameter

```
            training_param = SuperMLTrainingParam()
            training_param.median = True
            ml_repo.add(training_param, message='my first training parameter for my␣
↪own super ml algorithm')
```

Finally, we call `add_model` to define the overall model. Since our repo contains only one eval and train function as well as one unique training parameter we may call `add_model` without specifying them.

```
            ml_repo.add_model('my_model')
```

Important: The MLRepo's underlying RepoStore needs to store the objects of the model and training parameter classes. For this, the RepoStore calls the method `to_dict` to obtain a dictionary of the objects attributes which is then serialized. Here, one has to take care that all objects in this dictionary are serializable w.r.t. RepoStore's format. If your classes use simple standard python types, you may not need to adjust anything.

1.3 Tutorial

1.3.1 Repo initialization, training, evaluation

Creating a new repository

We first create a new repository for our task. The repository is the central key around all functionality is built. Similar to a repository used for source control in classical software development it contains all data and algorithms needed for the machine learning task. The repository needs storages for

- numerical objects such as arrays and matrices representing data, e.g. the input data or data from the valuation of the models

- small objects (of part of objects after cutting out the numerical objects), e.g. training parameter, model parameter.

To keep things simple, we may simply use the default constructor of the `MLRepo` creating in memory storages.

```
ml_repo = MLRepo(user='test_user')
```

Note that the memory interfaces used in this tutorial are useful for testing or playing around but may not be your choice for real life applications (except that you are willing to start your work again after your computer has been rebooted :-)). In this case, you could use a simple storage using json format to store small data in files while using a storage saving the numpy data in hdf5 files. In this case you have to specify this in a respective configuration dictionary.

```
config = {'user': 'test_user',
          'workspace': repo_path,
          'repo_store':
          {
              'type': 'disk_handler',
              'config': {
                  'folder': repo_path,
                  'file_format': 'pickle'
              }
          },
          'numpy_store':
          {
              'type': 'hdf_handler',
              'config': {
                  'folder': repo_path,
                  'version_files': True
              }
          },
          'job_runner':
          {
              'type': 'simple',
              'config': {
                  'throw_job_error': True
              }
          }
          }
ml_repo = MLRepo(user='test_user', config=config)
```

In addition to the storages the repository needs a reference to a `JobRunner` which the platform can use to execute different jobs needed during your ML development process. As long as we do not specify another `JobRunnner`, the `MLRepo` uses the most simple *`pailab.job_runner.job_runner.SimpleJobRunner`* as default, that executes everything sequential in the same thread the repository runs in. There are two possibilities to set the `JobRunner`. You may use the configuration settings as shown above. In this case, the `pailab.job_runner.job_runner_factory.JobRunnerFactory`is used to create the respective `JobRunner` within *MLRepo*'s constructor. Another possibility you may use (e.g. if you implemented your on JobRunner and you do not want to integrate it into the factory), you may simply instantiate the respective `JobRunner` and set it into the MLRepo's `job_runner` attribute

```
job_runner = SimpleJobRunner(None)
job_runner.set_repo(ml_repo)
ml_repo._job_runner = job_runner
```

Note: The ''MLRepo" uses the *`pailab.job_runner.job_runner.SimpleJobRunner`* as default, you do only have to set a `JobRunner` as shown above if you want to use a different one.

Adding training and test data

The data in the repository is handled by two different data objects:

- *pailab.ml_repo.repo_objects.RawData* is the object containing real data.
- pailab.ml_repo.repo.DataSet is the object containing the logical data, i.e. a reference to a RawData object together with a specification, which data from the RawData will be used. Here, one can specify a fixed version of the underlying RawData object (then changes to the RawData will not affect the derived DataSet) or a fixed or floating subset of the RawData by defining start and end index cutting the derived data just out of the original data.

Normally, for training and testing we will use DataSet. So, we first have to add the data in form of a RawData object and then define the respective DataSets based on this RawData.

Adding RawData

We first read the data from a csv file using pandas.

```
            import pandas as pd
            data = pd.read_csv('./examples/boston_housing/housing.csv')
```

Now data holds a pandas dataframe and we have to extract the respective x and y values as numpy matrices to use them to create the RawData object.

```
        input_variables = ['RM', 'LSTAT', 'PTRATIO']
        target_variables = ['MEDV']
        x = data.loc[:, input_variables].values
        y = data.loc[:, target_variables].values
```

Using the numpy objects we can now create the RawData object and add it to the repo.

```
        from pailab import RawData, RepoInfoKey

        raw_data = RawData(x, input_variables, y, target_variables, repo_info={
                           RepoInfoKey.NAME: 'raw_data/boston_housing'})
        ml_repo.add(raw_data)
```

Adding DataSet

Now, base on the RawData, we can add the training and test data sets.

```
        # create DataSet objects for training and test data
        training_data = DataSet('raw_data/boston_housing', 0, 300,
                                repo_info={RepoInfoKey.NAME: 'training_data',␣
↪RepoInfoKey.CATEGORY: MLObjectType.TRAINING_DATA})
        test_data = DataSet('raw_data/boston_housing', 301, None,
                            repo_info={RepoInfoKey.NAME: 'test_data',  RepoInfoKey.
↪CATEGORY: MLObjectType.TEST_DATA})
        # add the objects to the repository
        version_list = ml_repo.add(
            [training_data, test_data], message='add training and test data')
```

Note: We have to define the type of object via setting a value from :py:class:`pailab.repo.MLObjectType` for the `RepoInfoKey.CATEGORY` key object. The category is used by the MLRepo to support certain automatizations and checks.

The `version_list` variable is a dictionary that maps the object names of the added objects to their version.

Adding a model

The next step to do machine learning would be to define a model which will be used in the repository. A model consists of the following pieces

- a function for evaluating the model
- a function for model training
- a model parameter object holding the model parameters
- a training parameter object defining training parameters

We would like to use the DecisionTreeRegressor from sklearn in our example below. In this case, we do not have to define the pieces defined above, since pailab provides a simple interface to sklearn defined in the module *pailab.externals.sklearn_interface*. This interface provides a method *pailab.externals.sklearn_interface.add_model()* to add an arbitrary sklearn model as a model which can be handled by the repository. The method *pailab.externals.sklearn_interface.add_model()* creates internally a `pailab.repo_objects.Model` object defining the objects listed above and adds it to the repository. We refer to *Add a model* for details on setting up the model object and to :ref:‘integrating_model‘for details how to integrate your own algorithm or other external ml platforms.

```
        import pailab.externals.sklearn_interface as sklearn_interface
        from sklearn.tree import DecisionTreeRegressor
        sklearn_interface.add_model(
            ml_repo, DecisionTreeRegressor(), model_param={'max_depth': 5})
```

Train the model

Now, model training is very simple, since you have defined training and testing data as well as methods to value and fit your model and the model parameter. So, you can just call *pailab.ml_repo.repo.MLRepo.run_training()* on the repository, and the training is performed automatically. The training job is executed via the JobRunner you specified setting up the repository. All method of the repository involving jobs return the job id when adding the job to the JobRunner so that you can control the status of the task and see if it successfully finished.

```
        job_id = ml_repo.run_training()
```

The variable `job_id` contains a tuple with the id the job is stored in the repo and the respective version:

```
>> print(job_id)
('DecisionTreeRegressor/jobs/training', '69c7ce4a-512a-11e9-ab9f-fc084a6691eb')
```

This information can be used to retrieve the underlying job object. The job object contains certain useful information such as the status of the job, i.e. if it is waiting, running or if it has been finished, the time the job has been started or messages of errors that occurred during execution:

```
>>job = ml_repo.get(job_id[0], job_id[1])
>>print(job.state)
finished
>>print(job.started)
2100-03-28 08:23:41.668922
```

Note: The jobs are only executed if they have not yet been run on the input. So that if we call `run_training` again, we get a message that the job has already been run:

```
>> job_id = ml_repo.run_training()
>> print(job_id)
No new training started: A model has already been trained on the latest data.
```

We can check that the training was successful by checking whether a calibrated object for the specified model has been created. For this, we simply list all object names of objects from the category `MLObjectType.CALIBRATED_MODEL` stored within the repo using the :py:meth:`pailab.repo.MLRepo.get_names` method:

```
>>print(ml_repo.get_names(MLObjectType.CALIBRATED_MODEL))
['DecisionTreeRegressor/model']
```

As we see, an object with name `'DecisionTreeRegressor/model'` has been created and stored in the repo.

Model evaluation and error measurement

Evaluate a model

To measure errors and to provide plots the model must be evaluated on all test and training datasets. This can simply be accomplished by calling *pailab.ml_repo.repo.MLRepo.run_evaluation()*.

```
        job_id = ml_repo.run_evaluation()
```

This method has now applied the model's evaluation method to all test and training data stored in the repository and also stored the results. Similar to the model training we may list all results using the `get_names` method:

```
>>print(ml_repo.get_names(MLObjectType.EVAL_DATA))
['DecisionTreeRegressor/eval/sample2', 'DecisionTreeRegressor/eval/sample1']
```

As we see, we have two different objects containing the evaluation of the model, one for each dataset stored. Note that we can check what model and data has been used to create these evaluations. We just have to look at the `modification_info` attribute of the `repo_info` data attached to each object stored in the `MLRepo`:

```
>>eval_data = ml_repo.get('DecisionTreeRegressor/eval/sample2')
>>print(eval_data.repo_info.modification_info)
{'DecisionTreeRegressor/model': '69c86a46-512a-11e9-b7bd-fc084a6691eb',
'DecisionTreeRegressor': '687c5da8-512a-11e9-b0b4-fc084a6691eb',
'sample2': '6554763b-512a-11e9-938e-fc084a6691eb',
'eval_sklearn': '687bc058-512a-11e9-8b3e-fc084a6691eb',
'DecisionTreeRegressor/model_param': '687c5da7-512a-11e9-99c4-fc084a6691eb'}
```

The `modification_info` attribute is a dictionary that maps all objects involved in the creation of the respective object to their version that has been used to derive the object. We can directly see the versions of the calibrated model `'DecisionTreeRegressor/model'` as well as the version of the underlying data set `'sample2'`.

Define error measures

Now we may add certain error measures to the repository

```
        ml_repo.add_measure(MeasureConfiguration.MAX)
        ml_repo.add_measure(MeasureConfiguration.R2)
```

which can be evaluated by

```
        job_ids = ml_repo.run_measures()
```

As before, we get an overview of all measures computed and stored in the repository (as a repo object, see `pailab.ml_repo.repo_objects.measure`) using the `get_names` method:

```
>>print(ml_repo.get_names(MLObjectType.MEASURE))
['DecisionTreeRegressor/measure/test_data/max',
'DecisionTreeRegressor/measure/test_data/r2',
'DecisionTreeRegressor/measure/training_data/max',
'DecisionTreeRegressor/measure/training_data/r2']
```

Retrieving measures

The computed value is stored in the measurement object in the attribute value:

```
.. literalinclude:: ../../tests/tutorial_test.py
```

language python

start-after get measures

end-before end getting measures

prints the value of the measurement.

Creating a list of all objects

One can simply get an overview over all objects stored in the repository by calling *pailab.ml_repo.repo.MLRepo.get_names()* to retrieve a list of names of all objects of a specific category (see *pailab.ml_repo.repo.MLObjectType*). The following line will loop over all categories and print the names of all objects within this category contained in the repository.

1.3.2 Labeling, testing, consistency

Labeling model versions

The `MLRepo` offers the possibility to label a certain model version. This gives the user the possibility to mark certain models, e.g. labeling the model that goes into production or labeling the model which has the best error measure as a candidate for a future release. Labels are not only just nice to identify certain models more easily than remembering the version number, they are also supported by other methods form pailab: As we will see in this tutorial, consistency checks are applied to all labeled method, regression tests may be defined using labeled models or plotting methods will include the labeled models and explicitly highlight them.

Setting a label is quite simple using the `pailab.MLRepo.set_label()` method

```
        from pailab import LAST_VERSION
        ml_repo.set_label('prod', 'DecisionTreeRegressor/model',
                          model_version=LAST_VERSION, message='we found our first␣
↪production model')
```

Note that we have used the `LAST_VERSION` keyword. Instead of specifying the exact version string, nearly all methods who need a version do also accept the `LAST_VERSION` and `FIRST_VERSION` keywords.

The `set_label` method creates just an object of `pailab.repo_objects.Label` and stores it in the repository. Therefore listing all labels in the repo can be performed by using `get_names` again:

```
>>print(ml_repo.get_names(MLObjectType.LABEL))
['prod']
```

We can see what model and model version a label refers to by just getting the label object and checking the `name` and `version` attributes:

```
>>label = ml_repo.get('prod')
>>print(label.name)
>>print(label.version)
DecisionTreeRegressor/model
69c86a46-512a-11e9-b7bd-fc084a6691eb
```

Automated testing

There is a lot of debate whether unit testing or regression testing would make sense for ML. However, everyone should decide on his own for his project if it would make sense for his problems or not and pailab supports automated testing for those who want to apply it.

A test basically consists of two parts:

- A definition of the principal test containing the type of test and a definition for what data and models the tests are created,
- the tests itself which are also jobs executed by the `MLRepo`'s internal JobRunner.

Note: As a user, you normally just define the test using the respective *pailab.tools.tests.TestDefinition* and you do not instantiate an object from the *pailab.tools.tests.Test* class on your own.

Regression tests

We define a set of regression tests using *pailab.tools.tests.RegressionTestDefinition*. Here, pailab's RegressionTest compares specified error measures of a model with error measures of a reference model (typically the one in production, maybe labeled 'prod' ;-))

```
        import pailab.tools.tests
        reg_test = pailab.tools.tests.RegressionTestDefinition(
            reference='prod', models=None, data=None, labels=None,
            measures=[MeasureConfiguration.MAX],  tol=1000)
        reg_test.repo_info.name = 'reg_test'
        ml_repo.add(reg_test, message='regression test definition')
```

We may run the test by calling `pailab.ml_repo.MLRepo.run_test()`

```
        tests = ml_repo.run_tests()
```

where `tests` is a list of tuples, each containing the name of the test as well as the respective version:

```
>>print(tests)
[('DecisionTreeRegressor/tests/reg_test/test_data', '5b71ad5a-516f-11e9-bf7c-
↪fc084a6691eb'),
('DecisionTreeRegressor/tests/reg_test/training_data', '5b8b46ca-516f-11e9-990d-
↪fc084a6691eb')]
```

The attribute `result` of the test object contains the result of the test (if it was successful or not):

```
>>test = ml_repo.get('DecisionTreeRegressor/tests/reg_test/test_data')
>>print(test.result)
'succeeded'
```

Consistency checks

Pailab's `pailab.tools.checker` -submodule provides functionality to check for consistency and quality issues as well as for outstanding tasks (such as rerunning a training after the training set has been changed).

Model consistency

There are different checks to test model consistency such as if the tests of a model are up to date and succeeded or if the latest model is trained on the latest training data. All model tests are performed for labeled models and the latest model only.

The following checks are performed: - Is the latest model calibrated on the latest parameters and training data - Are all labeled models (including latest model) evaluated on the latest available training and test data - Are all measures of all labeled models computed on the latest data - Have all tests been run on the labeled models

```
        import pailab.tools.checker as checker
        inconsistencies = checker.run(ml_repo)
```

The variable `inconsistencies` contains a list of all inconsistencies found. In our case the list is currently empty since there are no inconsistencies:

```
>>print(inconsistencies)
[]
```

Now we change a model parameter but do not start a new training

```
        param = ml_repo.get('DecisionTreeRegressor/model_param')
        param.sklearn_params['max_depth'] = 2
        version = ml_repo.add(param)
```

We run the consistency check again:

```
>>print(inconsistencies)
[{'DecisionTreeRegressor/model:last': {'latest model version not on latest inputs':
{'DecisionTreeRegressor/model_param': {'modifier version': 'cdc3fed4-5192-11e9-a7fd-
↪fc084a6691eb',
        'latest version': 'cfe1b9fa-5192-11e9-b360-fc084a6691eb'}}}}]
```

Now we get a list containing one dictionary that contains the model inconsistencies. In our case, the dictionary shows one inconsistency: There are model inputs the latest calibrated model of `'DecisionTreeRegressor/model'` has not yet been calibrated on. It also shows us that the model parameter `'DecisionTreeRegressor/model_param'` is the input that is newer then the one used in the latest version.

We can fix this issue by running a new training:

```
>>ml_repo.run_training()
```

Rerun training fixes the training but leads to new problems. Now after having retrained, the evaluation of the new model on the data sets as well as the computation of the defined error measures are now missing:

```
>>print(inconsistencies)
[{'DecisionTreeRegressor/model:last': {
    'evaluations missing': {
        'training_data': '429f88ba-524d-11e9-98f6-fc084a6691eb',
        'test_data': '429f88ba-524d-11e9-98f6-fc084a6691eb'},
    'measures not calculated':
        {
        'DecisionTreeRegressor/measure/training_data/max',
        'DecisionTreeRegressor/measure/test_data/r2',
        'DecisionTreeRegressor/measure/test_data/max',
        'DecisionTreeRegressor/measure/training_data/r2'}}}]
```

Now we may fix these issues by calling first `pailab.repo.MLRepo.run_evaluation()` and then `pailab.repo.MLRepo.run_measures()` or we can simply call `run_evaluation` only, setting the parameter `run_descendants` to `True`. By doing so, the `MLRepo` resolves all steps of the build pipeline following the model evaluation

>>print(checker.run(ml_repo)) []

Training and test data consistency

pailab does also perform checks w.r.t. the training and test data. Here, one check is if test and training data overlap. To illustrate this, we add a second test data set to the repo which overlaps with the training data. Note that we first run the evaluation on the new data set so that we do not see again the errors that evaluation or error measures are missing for this data

```
        test_data_2 = DataSet('raw_data/boston_housing', 0, 50,
                              repo_info={RepoInfoKey.NAME: 'test_data_2',
                                         RepoInfoKey.CATEGORY: MLObjectType.TEST_DATA}
                              )
        ml_repo.add(test_data_2)
        ml_repo.run_evaluation(run_descendants=True)
```

Now, performing the check shows a lot of inconsistencies a check:

```
>>print(checker.run(ml_repo))
[{'test_data_2': {'training and test data overlap': {'test_data_2': 'ecdc36ee-5465-
↪11e9-92e2-fc084a6691eb', 'training_data': 'e6ac4eba-5465-11e9-b956-fc084a6691eb'}}}]
```

Tests

We may also check the overall test status. Here we have to call `pailab.tools.checker.Tests.run()`:

```
>>print(checker.Tests.run(ml_repo))
{'DecisionTreeRegressor/model:323c05e8-5483-11e9-88ea-fc084a6691eb':
    {'DecisionTreeRegressor/tests/reg_test/test_data':
        'Test for model DecisionTreeRegressor/model,
        version 323c05e8-5483-11e9-88ea-fc084a6691eb on latest data test_data missing.
↪',
    'DecisionTreeRegressor/tests/reg_test/test_data_2':
        'Test for model DecisionTreeRegressor/model, version 323c05e8-5483-11e9-88ea-
↪fc084a6691eb on latest data test_data_2 missing.',
    'DecisionTreeRegressor/tests/reg_test/training_data':
        'Test for model DecisionTreeRegressor/model, version 323c05e8-5483-11e9-88ea-
↪fc084a6691eb on latest data training_data missing.'
    },
'DecisionTreeRegressor/model:prod':
    {'DecisionTreeRegressor/tests/reg_test/test_data_2':
        'Test for model DecisionTreeRegressor/model, version 301bcc1e-5483-11e9-82a2-
↪fc084a6691eb on latest data test_data_2 missing.'
    }
}
```

We see that the latest model as well as the *'prod'* labeled model gives us some errors since the regression tests have not ben run on all data sets. We can fix these messages by simply calling `run_tests`:

```
>>ml_repo.run_tests()
>>print(checker.Tests.run(ml_repo))
{}
```

1.4 API Reference

1.4.1 ml_repo

class MLRepo(*workspace=None*, *user=None*, *config=None*, *save_config=False*)

Repository for doing machine learning

The repository and his extensions provide a solid fundament to do machine learning science supporting features such as:

- auditing/versioning of the data, models and tests
- best practice standardized plotting for investigating model performance and model behaviour
- automated quality checks

Parameters

- **workspace** (*[type]*) – [description]. Defaults to None.
- **user** (*str*) – the user. Defaults to None.
- **config** (*dict*) – the configuration to use. Defaults to None.
- **save_config** (*bool*) – determines whether to save the configuration or not. Defaults to False.

add(*repo_object*, *message=''*, *category=None*)

Add a repo_object or list of repo objects to the repository.

Raises an exception if the category of the object is not defined in the object and if it is not defined with the category argument. It raises an exception if an object with this id does already exist.

Parameters

- **repo_object** (RepoObject) – repo_object or list of repo_objects to be added, will be modified so that it contains the version number
- **message** (*str*) – commit message. Defaults to ‘’.
- **category** (MLObjectType) – Category of repo_object which overwrites the objects category.. Defaults to None.

Returns str or dictionary – version number of object added or dictionary of names and versions of objects added

add_eval_function(*f*, *repo_name=None*)
Add the function to evaluate the model

Parameters

- **module_name** (*str*) – module where function is located
- **function_name** (*str*) – function name
- **repo_name** (*str*) – identifier of the repo object used to store the information, if None, the name is set to module_name.function_name. Defaults to None.

add_measure(*measure*, *coordinates=None*)
Add a measure to the repository

If the measure already exists, it returns the message

Parameters

- **measure** (*str*) – string defining the measure, i.e MAX,...
- **coordinates** (*list of str*) – list of strings defining the coordinates (by name) used for the measure, if None, all coordinates will be used. Defaults to None.

add_model(*model_name*, *model_eval=None*, *model_training=None*, *model_param=None*, *training_param=None*, *preprocessors=None*)
Add a new model to the repo

Parameters

- **model_name** (*str*) – identifier of the model
- **model_eval** (*str*) – identifier of the evaluation function in the repo to evaluate the model, if None and there is only one evaluation function in the repo, this function will be used
- **model_training** (*str*) – identifier of the training function in the repo to train the model, if None and there is only one evaluation function in the repo, this function will be used
- **model_param** (*str*) – identifier of the model parameter in the repo, if None and there is exactly one ModelParameter in teh repo, this will be used,. Defaults to None. otherwise it is assumed that no model_params are needed
- **training_param** (*str*) – identifier of the training parameter, if None and there is only one training_parameter object in the repo, . Defaults to None. this will be used. If an empty string is given as training parameter, we assume that the algorithm does not need a training pram.

- **preprocessors** (*list*) – list of preprocessors to be execute. Defaults to None. this is a list of strings

add_preprocessing_fitting_function(*f*, *repo_name=None*)
Add function to fit a preprocessor

Parameters

- **module_name** (*str*) – module where function is located
- **function_name** (*str*) – function name
- **repo_name** (*tring*) – identifier of the repo object used to store the information, if None, the name is set to module_name.function_name. Defaults to None.

add_preprocessing_transforming_function(*f*, *repo_name=None*)
Add function to transform the data by a preprocessor

Parameters

- **module_name** (*str*) – module where function is located
- **function_name** (*str*) – function name
- **repo_name** (*str*) – identifier of the repo object used to store the information, if None, the name is set to module_name.function_name. Defaults to None.

add_preprocessor(*preprocessor_name*, *transforming_function=None*, *fitting_function=None*, *preprocessor_param=None*)
Add a new preprocessor to the repo

Parameters

- **preprocessor_name** (*str*) – identifier of the preprocessor
- **transforming_function** (*str*) – identifier of the transforming function in the repo, if None and there is only one transforming function in the repo, this function will be used
- **fitting_function** (*str*) – identifier of the fitting function in the repo to fit the preprocessor, if None the preprocessor does not need to be fitted
- **preprocessor_param** (*str*) – identifier of the preprocessor parameter. Defaults to None.

Raises Exception – Raises an error if the preprocessing transforming function is not in repo

add_raw_data(*name*, *data*, *input_names=None*, *data_y=None*, *target_names=None*, *file_format=None*, *axis=1*)
Adds a RawData object to the repository.

This methods creates/reads from the given data/file a RawData object and adds it to the repository.

Examples

Read data from csv file 'test_data.csv' and use columns with headers 'x0', 'x1' as input data and column with label 'x2' as target, store the results under name 'my_data':

```
>>ml_repo.add_raw_data('my_data', 'test_data.csv', ['x0', 'x1], file_format =
↪'csv')
```

Create data from a DataFrame test where the columns 'x0', 'x1' are used as input and no target is specified:

```
>>ml_repo.add_raw_data('my_data', test, ['x0', 'x1])
```

Parameters

- **name** (*str*) – Name of RawData in repository (if name does not start with 'raw_data/' this is added.
- **data** (*str, numpy ndarray or pandas DataFrame*) – Eithr a pandas DataFarme, a numpy ndarray or a string that is interpreted as filename of the underling data.
- **input_names** (*iterable of str, optional*) – List of the input variables names. Defaults to None.
- **data_y** (*str or numpy ndarray, optional*) – Either a numpy ndarray or a string defining the filename of th y-data (not valid if file_format=='csv'). Defaults to None.
- **target_names** (*iterable of str, optional*) – List of the target variables names. Defaults to None.
- **file_format** (*'csv' or 'numpy', optional*) – File type which can be either csv or numpy (numpy means an ndarray stored with numpy.save). Defaults to None.
- **axis** (*int, optional*) – If only an ndarray is given but target variables are defined, this array will b split into weo arrays (one for input, one for target) along this axis. Defaults to 1.

Returns version number of RawData object added

Return type str

add_test_data(*name*, *raw_data_name*, *start_index=0*, *end_index=None*, *raw_data_version='last'*)

Add test data as a DataSet to the repository.

This method defines a DataSet and adds it to the repository. A DataSet is a logical unit based on a RawData object and defines the range of data that is taken from the respective RawData data.

Parameters

- **name** (*str*) – Name of respective object in repository.
- **raw_data_name** (*str*) – Name of the underlying RawData object that is used as basis.
- **start_index** (*int, optional*) – Start index where test data starts from underlying RawData. Defaults to 0.
- **end_index** (*int, optional*) – End index where test data end. Defaults to None.
- **raw_data_version** (*str*) – Version of underlying RawData (if 'last', always the latest RawData will be used to derive the respective DataSet). Defaults to 'last'

add_training_data(*name*, *raw_data_name*, *start_index=0*, *end_index=None*, *raw_data_version='last'*)

Add training data as a DataSet to the repository.

This method defines a DataSet and adds it to the repository. A DataSet is a logical unit based on a RawData object and defines the range of data that is taken from the respective RawData data.

Parameters

- **name** (*str*) – Name of respective object in repository.
- **raw_data_name** (*str*) – Name of the underlying RawData object that is used as basis.
- **start_index** (*int, optional*) – Start index where training data starts from underlying RawData. Defaults to 0.

- **end_index** (*int, optional*) – End index where training data end. Defaults to None.
- **raw_data_version** (*str*) – Version of underlying RawData (if 'last', always the latest RawData will be used to derive the respective DataSet). Defaults to 'last'

add_training_function(*f*, *repo_name=None*)
Add function to train a model

Parameters

- **module_name** (*str*) – module where function is located
- **function_name** (*str*) – function name
- **repo_name** (*tring*) – identifier of the repo object used to store the information, if None, the name is set to module_name.function_name. Defaults to None.

delete(*name*, *version*)
Delete a specific object.

It deletes the object. If other objects were modified by this object, it throws an exception that first the modified objects must be deleted.

Parameters

- **name** (*str*) – name of the object
- **version** (*str*) – version of the object

Raises Exception – If the object has depending objects, it can not be deleted and an error is thrown.

get(*name*, *version='last'*, *full_object=False*, *modifier_versions=None*, *obj_fields=None*, *repo_info_fields=None*, *throw_error_not_exist=True*, *throw_error_not_unique=True*)
Get repo objects. It throws an exception, if an object with the name does not exist.

Parameters

- **name** (*str*) – the object name
- **version** (*str*) – object version, default is latest (-1). If the fields are nested (an element of a dictionary which is an element of a dictionary, use path notation to the element, i.e. p/elem1/elem2 to get p[elem1][elem2]). Defaults to repo_store.RepoStore.LAST_VERSION.
- **full_object** (*bool*) – flag to determine whether the numpy objects are loaded (True->load). Defaults to False.
- **modifier_versions** (*[type]*) – [description]. Defaults to None.
- **obj_fields** (*[type]*) – [description]. Defaults to None.
- **repo_info_fields** (*[type]*) – [description]. Defaults to None.
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.
- **throw_error_not_unique** (*bool*) – true - throw error if item is not unique, else return []. Defaults to True.

Raises Exception – raises an exception if no object with the specific name is found

Returns RepoObject or list thereof – The repo object

static get_calibrated_model_name(*model_name*)
For a model name the calibrated model name is returned

Parameters **model_name** (*str*) – model name

Returns string – the calibrated model name

get_commits(*version_start='first'*, *version_end='last'*)
gets the commits

Parameters

- **version_start** (*str*) – only display versions after version_start. Defaults to repo_store.RepoStore.FIRST_VERSION.
- **version_end** (*str*) – only display versions up to version_end. Defaults to repo_store.RepoStore.LAST_VERSION.

Returns list of commit infos – returns a list of commit infots

static get_eval_name(*model*, *data*)
Return name of the object containing evaluation results

Parameters

- **model** (*ModelDefinition object or str*) –
- **{RawData or DataSet object or str} --** (*data*) –

Returns string – name of valuation results

get_history(*name*, *repo_info_fields=None*, *obj_member_fields=None*, *version_start='first'*, *version_end='last'*)
Return a list of histories of object member variables without bigobjects

Parameters

- **name** (*str*) – the object name
- **repo_info_fields** (*list of strings*) – List of fields from repo_info which will be returned in the dictionary. If List contains flag 'ALL', all fields will be returned.. Defaults to None.
- **obj_member_fields** (*list of strings*) – List of member atributes from repo_object which will be returned in the dictionary. If List contains flag 'ALL', all attributes will be returned.. Defaults to None.
- **version_start** (*str*) – only display versions after version_start. Defaults to repo_store.RepoStore.FIRST_VERSION.
- **version_end** (*str*) – only display versions up to version_end. Defaults to repo_store.RepoStore.LAST_VERSION.

Returns str or list of strings – returns a list of the objects

get_ml_repo_store()
Return the storage for the ml repo

Returns RepoStore – the storage for the RepoObjects

get_names(*ml_obj_type*)
Get the list of names of all repo_objects from a given repo_object_type in the repository.

Parameters **ml_obj_type** (MLObjectType) – MLObjectType specifying the types of objects names are returned for.

Returns list of strings – list of object names for the given category.

get_numpy_data_store()
Return the numpy data store of the ml repo

Returns numpy_handler – the numpy repo

get_training_data(*version='last'*, *full_object=True*, *model=None*, *model_version='last'*)
Returns training data for a model.

It returns the training data in the repo for a specified model. If there is only one set of training data in the repo, this set will be returned. Otherwise, the model is loaded and the training data is used as defined in the model. If in this case a model is not specified the method throws an exception.

Parameters

- **version** (*str*) – version of data object. Defaults to repo_store.RepoStore.LAST_VERSION.
- **full_object** (*bool*) – if True, the complete data is returned including numpy data. Defaults to True.
- **model** (*str*) – Name of model definition for which the training data will be returned.
- **model_version** (*str*) – Version of model definition for which teh trainin data will be returned.

pull()
Pull changes from an external repo

push()
Push changes to an external repo.

run(*job*)
Executes a job

Parameters **job** (*Job*) – The job object to be executed

Returns [type] – Return the name and version of the job or a message that the job does not need to be rerun

run_evaluation(*model=None*, *message=None*, *model_version='last'*, *datasets={}*, *predecessors=[]*, *run_descendants=False*, *labels=None*)
Evaluate the model on all datasets.

Parameters

- **model** (*str*) – name of model to evaluate, if None and only one model exists. Defaults to None.
- **message** (*str*) – message inserted into commit, if None: an automated message is created. Defaults to None.
- **model_version** (*str*) – version of model to be evaluated.. Defaults to repo_store.RepoStore.LAST_VERSION.
- **datasets** (*dict*) – dictionary of datasets (names and version numbers) on which the model is evaluated. . Defaults to {}.
- **predecessors** (*list*) – list of jobs which shall have been completed successfull before the evaluation is started. Default is all datasets from testdata on latest version.. Defaults to [].
- **run_descendants** (*bool*) – if True also run all decendant jobs. Defaults to False.
- **labels** (*[type]*) – [description]. Defaults to None.

Returns list of strings – a list of the job ids

run_measures(*model=None*, *message=None*, *model_version='last'*, *datasets={}*, *measures={}*, *predecessors=[]*, *labels=None*)

Run the measures

Parameters

- **model** (*str*) – name of model to evaluate, if None and only one model exists. Defaults to None.
- **message** (*str*) – message inserted into commit, if None: an automated message is created. Defaults to None.
- **model_version** (*str*) – version of model to be evaluated.. Defaults to repo_store.RepoStore.LAST_VERSION.
- **datasets** (*dict*) – dictionary of datasets (names and version numbers) on which the model is evaluated. . Defaults to {}.
- **predecessors** (*list*) – list of jobs which shall have been completed successfull before the evaluation is started. Default is all datasets from testdata on latest version.. Defaults to [].
- **run_descendants** (*bool*) – if True also run all decendant jobs. Defaults to False.
- **labels** (*[type]*) – [description]. Defaults to None.

Returns list of strings – a list of the job ids

run_tests(*test_definitions=None*, *predecessors=[]*)

Run tests for a specific model version.

Parameters

- **test_definitions** (*list or set*) – List or set of names of the test definitions which shall be executed. If None, all test definitions are executed.. Defaults to None.
- **predecessors** (*list*) – list of jobs which shall have been completed successfull before the evaluation is started. Default is all datasets from testdata on latest version.. Defaults to [].

Returns str – ticket number of job

run_training(*model=None*, *message=None*, *model_version='last'*, *training_function_version='last'*, *training_data_version='last'*, *training_param_version='last'*, *model_param_version='last'*, *run_descendants=False*)

Run the training algorithm.

Parameters

- **model** (*str*) – the identifyer of the model. Defaults to None.
- **message** (*str*) – the commit message. Defaults to None.
- **model_version** (*str*) – the version of the model. Defaults to repo_store.RepoStore.LAST_VERSION.
- **training_function_version** (*str*) – the version of the training function. Defaults to repo_store.RepoStore.LAST_VERSION.
- **training_data_version** (*str*) – the version of the training data. Defaults to repo_store.RepoStore.LAST_VERSION.
- **training_param_version** (*str*) – the version of the training parameter. Defaults to repo_store.RepoStore.LAST_VERSION.

- **{str} --the version of the model parameter. Defaults to repo_store.RepoStore.LAST_VERSION.** (*model_param_version*) –
- **run_descendants** (*bool*) – if True also run all decendant jobs. Defaults to False.

Returns [type] – return name and version or message

set_label(*label_name*, *model=None*, *model_version='last'*, *message=''*)
Label a certain model version.

It checks if a model with this version really exists and throws an exception if such a model does not exist. This method labels a certain model version.

Parameters

- **label_name** (*str*) – the label name
- **model** (*str*) – the identifyer of the model. Defaults to None.
- **model_version** (*str*) – model version for which the label is set.. Defaults to repo_store.RepoStore.LAST_VERSION.
- **message** (*str*) – commit message. Defaults to ''.

1.4.2 repo_objects

This module contains a bunch of different RepoObjects.

In principal, all objects that can be stored within pailab's MLRepo are called a RepoObject. So, if you need a new object apart from those documented here, you just have to implement the respective interfaces, so that the object can be processed by pailab. This may be accomplished in three different ways:

- Inherit your class from the `pailab.repo_objects.RepoObject` class. This may not be very pythonic, but it easily shows you which interfaces you definitively have to implement.
- If you have a very simple object you may use the decorator `pailab.repo_objects.repo_object_init` in conjunction with your classe's constructor to make your class a RepoObject.
- Just implement the methods needed (again look at `pailab.repo_objects.RepoObject` to what has to be defined).

class CommitInfo(*message*, *author*, *objects*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Stores each commit including the commit message and the objects commited.

:param : param message (string): commit message :param : param author (string): author :param objects: dictionary of names of committed objects and version numbers :type objects: dictionary

class DataSet(*raw_data*, *start_index=0*, *end_index=None*, *raw_data_version='last'*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Class used to define data used e.g. for training or testing.

This class refers to some RawData object and a start- and endindex. The repository

Parameters

- **raw_data** (*str*) – id of raw_data the dataset refers to
- **start_index** (*int*) – index of first entry of the raw data used in the dataset. Defaults to 0.
- **end_index** (*int or None*) – end_index of last entry of the raw data used in the dataset (if None, all including last element are used). Defaults to None.

- **raw_data_version** (*str*) – version of RawData object the DataSet refers to. Defaults to 'last'.
- **repo_info** (RepoInfo) – dictionary of the repo info}). Defaults to RepoInfo().

Raises Exception – raises an exception if the start index is after the end index

set_data(*raw_data*)
Set the data from the given raw_data.

Parameters **raw_data** (RawData) – the raw data used to set the data from

Raises Exception – if end_index id less than start_index

class Function(*f*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)

create()
Returns the function object

Returns the function object

Return type function object

get_version()
returns the version

Returns [type] – the module version

class Label(*model_name*, *model_version*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
RepoObject to label a certain model version

class Measure(*value*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
the measure repo object

class MeasureConfiguration(*measures*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
RepoObject defining a configuration for all measures which shall be computed.

L2 = 'l2'

MAX = 'max'

MSE = 'mse'

R2 = 'r2'

add_measure(*measure*, *coords=None*)
add a measure to the repo object

Parameters

- **measure** (*[type]*) – the measure
- **coords** (*[type]*) – the coordinates. Defaults to None.

static get_name(*measure_def*)
function to return a name of the measure

Parameters **measure_def** (MeasureConfiguration) – the measure definition

Returns str – the name of the measure

class Model(*preprocessors=None*, *eval_function=None*, *train_function=None*, *train_param=None*, *model_param=None*, *training_data=None*, *test_data=None*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)

get_test_data(*ml_repo*)
Returns all test data in the repo relevant for this model.

Parameters **ml_repo** (MLRepo) – The repository from which the test data is taken

Returns list of names of the test data that applied to this model

class Preprocessor(*transforming_function*, *fitting_function=None*, *preprocessing_param=None*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Preprocessor class

class RawData(*x_data*, *x_coord_names*, *y_data=None*, *y_coord_names=None*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Class to store numpy data.

class RepoInfo(*kwargs={}*, *name=None*, *version=None*, *category=None*, *modification_info=None*)
Contains all repo relevent information

This class contains all repo relevant information such as version, name, descriptions. It must be a member of all objects which are handled by the repo.

get_dictionary()
Return repo info as dictionary

set_fields(*kwargs*)
Set repo info fields from a dictionary

Parameters **kwargs** (*dict*) – additional arguments

class RepoInfoKey
Enums to describe all possible repository informations.

AUTHOR = 'author'

BIG_OBJECTS = 'big_objects'

CATEGORY = 'category'

CLASSNAME = 'classname'

COMMIT_DATE = 'commit_date'

COMMIT_MESSAGE = 'commit_message'

DESCRIPTION = 'description'

MODIFICATION_INFO = 'modification_info'

NAME = 'name'

VERSION = 'version'

class RepoObject(*repo_info*)
Base class for objects which are handled b the repository.

from_dict(*repo_obj_dict*)
set object from a dictionary

Parameters **repo_object_dict** (*dict*) – dictionary with the object data

numpy_from_dict(*repo_numpy_dict*)
sets the attributes of the numpy dictionary

Parameters **repo_numpy_dict** (*dict*) – dictionary with the object data

numpy_to_dict()
function to get the attributes as a dictionary

Returns dict – dictionary of the attributes

to_dict()
Return a data dictionary for a given repo_object without the big data objects

Returns dict – dictionary of data

class Result(*data*, *big_data=None*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
the result repo object

numpy_from_dict(*repo_numpy_dict*)
sets the attributes of the numpy dictionary

Parameters **repo_numpy_dict** (*dict*) – dictionary with the object data

numpy_to_dict()
returns the big data object

Returns [type] – the big data

create_repo_obj(*obj*)
Create a repo_object from a dictionary.

This function creates a repo object from a dictionary in a factory-like fashion. It uses the obj['repo_info']['classname'] within the dictionary and constructs the class using get_object_from_classname. It throws an exception if the dictionary does not contain an 'repo_info' key.

Parameters **obj** (*dict*) – dictionary containing all informations for a repo_object.

Raises Exception – raises an exception if the dictionary is not a repo dictionary

Returns [type] – the object of the specified class

create_repo_obj_dict(*obj*)
Create from a repo_object a dictionary with all values to handle the object within the repo

Parameters **obj** (RepoObject) – repository object

Returns dict – returns the dictionary

get_object_from_classname(*classname*, *data*)
Returns an object instance for given classname and data dictionary.

Parameters

- **classname** (*str*) – Full classname as string including the modules, e.g. repo.Y if class Y is defined in module repo.
- **data** (*dict*) – dictionary of data used to initialize the object instance.

Returns [type] – Instance object of class.

class repo_object_init(*big_objects=[]*)
Decorator class to modify a constructor so that the class can be used within the ml repository as repo_object.

from_dict(*repo_obj_dict*)
set object from a dictionary

Parameters

- **repo_object** (RepoObject) – repo_object which will be set from the dictionary
- **repo_object_dict** (*dict*) – dictionary with the object data

init_repo_object(*init_self*, *repo_info*)
initialiser for repo objects

Parameters

- **init_self** (*[type]*) – [description]
- **repo_info** (*dict*) – the repository info

numpy_from_dict(*repo_numpy_dict*)
function to transform a dictionary to a numpy

Parameters

- **repo_obj** (RepoObject) – the repo object
- **repo_numpy_dict** (*numpy dict*) – the repo numpy dictionary

numpy_to_dict()
function to get the attributes as a dictionary

Parameters **repo_obj** (RepoObject) – the repo object

Returns dict – dictionary of the attributes

to_dict()
Return a data dictionary for a given repo_object

Parameters **repo_obj** (RepoObject) – A repo_object, i.e. object which provides the repo_object interface

Returns dict – dictionary of data

object types

class MLObjectType
Enum describing all ml object types.

The MLObjectType is assigned to each object in the MLRepo. It is used to structure all objects and to support consistency checks and automatic pipelines, the following types are defined:

- EVAL_DATA: evaluation data (result from evaluation of a model)
- RAW_DATA: raw data, i.e. simple numpy structures most often used to derive test or training data from the RawData
- TRAINING_DATA: training data used for model training
- TEST_DATA: data used for model testing
- TEST: concrete test
- TEST_DEFINITION: definition of a test which is applied to respective data and models to obtain a test
- MODEL_PARAM: model parameter
- TRAINING_PARAM: training parameter
- TRAINING_FUNCTION: function to train e certain model
- MODEL_EVAL_FUNCTION: function to evaluate a certain model
- PREPROCESSOR_PARAM: preprocessing parameter
- PREPROCESSOR: definition of a preprocessor
- PREPROCESSING_FITTING_FUNCTION: function to fit the preprocessor
- PREPROCESSING_TRANSFORMING_FUNCTION: function to apply preprocessing to data

- LABEL: model label
- MODEL: definition of a model
- CALIBRATED_MODEL: object containing a calibrated instande of a model
- COMMIT_INFO: internally used to store commit messages
- MAPPING: internally used mapping object to map an object's name to the object's category
- MEASURE: computed measure (e.g. norm of error)
- MEASURE_CONFIGURATION: the configuration of all measures applied to the model
- RESULT: object holding results
- JOB: a job
- TRAINING_STATISTIC: object holding training statistics, e.g. training history
- CACHED_VALUE: cached return values of time consuming functions

1.4.3 repo stores

Base classes

class NumpyStore
class to handle big objects

add(*name*, *version*, *numpy_dict*)
Add numpy data from an object to the storage.

Parameters

- **name** (*str*) – Name (as string) of object
- **version** (*str*) – object version
- **numpy_dict** (*numpy dict*) – numpy dictionary

append(*name*, *version_old*, *version_new*, *numpy_dict*)
Append data to an existing object

Parameters

- **name** (*str*) – name of data object to be returned
- **version_old** (*str*) – version of the object where the data will be appended
- **version_new** (*str*) – version of the new objct after appending the data
- **numpy_dict** (*dict*) – dictionary containing the values

get(*name*, *version*, *from_index=0*, *to_index=None*)
get the numpy object for a name and a version, rows can be used

Parameters

- **name** (*str*) – identifier of the object
- **version** (*str*) – version of the object
- **from_index** (*int*) – the index from which the data should be taken. Defaults to 0.
- **to_index** (*int or None*) – the index to which the data is returned (None means till the end). Defaults to None.

Returns numpy array – the numpy object to return

pull()
Pull changes from an external repo

push()
Push changes to an external repo.

class RepoScriptStore

add(*script_file*)
Add a script to the storage.

Parameters **script_file** (*string*) – file (incluing path) of script

get(*name*, *versions=None*)

class RepoStore

LAST_VERSION = 'last'
Abstract base class for all storages which can be used in the ML repository

add(*obj*)
Add an object to the storage.

Parameters **obj** (*RepoObject | list(RepoObject)*) – repository object or list o repository objects

Raises Exception if an object with same name already exists.

get(*name*, *versions=None*, *modifier_versions=None*, *obj_fields=None*, *repo_info_fields=None*, *throw_error_not_exist=True*, *throw_error_not_unique=True*)
Get a dictionary/list of dictionaries fulffilling the conditions.

Returns a list of objects matching the name and whose -version is in the given list of versions -modifiers match the version number/are in the list of version numbers of the given modifiers

Parameters

- **name** (*str*) – object id
- **versions** (*list, version_number, tuple*) – either a list of versions or a single version of the objects to be returned,. Defaults to None. if None, the condition on version is ignored. If a tuple is given, this tuple defines a version intervall, i.e. all versions between the first and last entry (both including) are returned. In addition FIRST_VERSION and LAST_VERSION can be used for versions to access the last/first version.
- **modifier_versions** (*dictionary*) – modifier ids together with version specs which are matched by the returned object.. Defaults to None.
- **obj_fields** (*list of str or str*) – list of strings identifying the fields which will be returned in the dictionary, if None, no fields are returned, if set to 'all', all fields will be returned . Defaults to None.
- **repo_info_fields** (*list of str or str*) – list of strings identifying the fields of the repo_info dict which will be returned in the dictionary, if None, no fields are returned, if set to 'all', all fields will be returned. Defaults to None.
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

- **throw_error_not_unique** (*bool*) – true - throw error if item is not unique, else return []. Defaults to True.

Returns RepoObject or list thereof – The repo object

get_first_version(*name*, *throw_error_not_exist=True*)
Return version number of first (in a temporal sense) object in storage

Parameters

- **name** (*str*) – object name for which the version is returned
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises NotImplementedError – [description]

get_latest_version(*name*, *throw_error_not_exist=True*)
Return latest version number of object in the storage

Parameters

- **name** (*str*) – object name
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Returns str – latest version number

get_names(*ml_obj_type*)
Return the names of all objects belonging to the given category.

Parameters **ml_obj_type** (*str*) – Value of MLObjectType-Enum specifying the category for which all names will be returned

get_version(*name*, *offset*, *throw_error_not_exist=True*)
Return versionnumber for the given offset

If offset >= 0 it returns the version number of the offset version, if <0 it returns according to the python list logic the version number of the (offset-1)-last version

Parameters

- **name** (*str*) – name of object
- **offset** (*int*) – offset
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

object_exists(*name*, *version='last'*)
Returns True if an object with the given name and version exists.

Parameters

- **name** (*string*) – object name
- **version** (*version number*) – version number. Defaults to LAST_VERSION.

pull()
Pull changes from an external repo

push()
Push changes to an external repo.

replace(*obj*)
Overwrite existing object without incrementing version

Parameters **obj** (RepoObject) – repo object to be overwritten

Memory storages

class NumpyMemoryStorage
Bases: *pailab.ml_repo.repo_store.NumpyStore*

add(*name*, *version*, *numpy_dict*)
Add numpy data from an object to the storage.

Parameters

- **name** (*str*) – identifier (as string) of object
- **version** (*str*) – object version
- **numpy_dict** (*numpy dict*) – numpy dictionary

append(*name*, *version_old*, *version_new*, *numpy_dict*)
appends an numpy dictionary to an existing object

Parameters

- **name** (*str*) – identifier of the object
- **version_old** (*str*) – the old version of the object
- **version_new** (*str*) – the new version of the object
- **numpy_dict** (*numpy dict*) – the numpy dictionary to append

Raises Exception – raises an exception if the object does not exist

get(*name*, *version*, *from_index=0*, *to_index=None*)
get the numpy object for a name and a version, rows can be used

Parameters

- **name** (*str*) – identifier of the object
- **version** (*str*) – version of the object
- **from_index** (*int*) – the index from which the data should be taken. Defaults to 0.
- **to_index** (*int or None*) – the index to which the data is returned (None means till the end). Defaults to None.

Raises

- Exception – raises an exception if no object with the name exists
- Exception – raises an exception if no object and with the version exists

Returns numpy array – the numpy object to return

class RepoObjectMemoryStorage
Bases: *pailab.ml_repo.repo_store.RepoStore*

The repo object memory storage. This class is used to store repo object (excluding large objects) in the memory. The importance of the handler is mostly for testing purposes.

get_first_version(*name*, *throw_error_not_exist=True*)
Determine the first version of the object

Parameters

- **name** (*str*) – identifier of the object
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises Exception – Raises an exception if the object does not exists

Returns str – the first version string of the object

get_latest_version(*name*, *throw_error_not_exist=True*)
Determine the latest version of the object

Parameters

- **name** (*str*) – identifier of the object
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises Exception – Raises an exception if the object does not exists

Returns str – the latest version string of the object

get_names(*category*)
Return the names of all objects belonging to the given category.

Parameters **ml_obj_type** (*str*) – Value of MLObjectType-Enum specifying the category for which all names will be returned

Returns list of str – a list of all objects in the category

get_version(*name*, *offset*, *throw_error_not_exist=True*)
Return the newest version up to offset versions

Parameters

- **name** (*str*) – the identifier of the object
- **offset** (*int*) – the offset
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises

- Exception – raises an error if the offset is higher than the number of versions available
- Exception – raises an exception if the object does not exists and throw_error_not_exist == True

Returns str – the version

replace(*obj*)
Overwrite existing object without incrementing version

Parameters **obj** (RepoObject) – repo object to be overwritten

RepoObjectDiskStorage

class RepoObjectDiskStorage(*folder*, *file_format='pickle'*)
The RepoObjectDiskStorage class

check_integrity()
Checks if files are missing or have not yet been added

Returns dictionary – contains sets of missing files and/or set of files not yet added

close_connection()
Closes the database connection

get_config()
return the configuration

Returns dict – a dictionary of the configuration

get_first_version(*name*, *throw_error_not_exist=True*)
Determine the first version of the object

Parameters

- **name** (*str*) – identifier of the object
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises Exception – Raises an exception if the object does not exists

Returns str – the first version string of the object

get_latest_version(*name*, *throw_error_not_exist=True*)
Determine the latest version of the object

Parameters

- **name** (*str*) – identifier of the object
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises Exception – Raises an exception if the object does not exists

Returns str – the latest version string of the object

get_names(*ml_obj_type*)
Return the names of all objects belonging to the given category.

Parameters **ml_obj_type** (*str*) – Value of MLObjectType-Enum specifying the category for which all names will be returned

Returns list of str – a list of all objects in the category

get_version(*name*, *offset*, *throw_error_not_exist=True*)
Return the newest version up to offset versions

Parameters

- **name** (*str*) – the identifier of the object
- **offset** (*int*) – the offset
- **throw_error_not_exist** (*bool*) – true - throw error if not exists, else return []. Defaults to True.

Raises Exception – raises an exception if the object does not exists and throw_error_not_exist == True

Returns str – the version

get_version_condition(*name*, *versions*, *version_column*, *time_column*)
returns the condition part of the versions for the sql statement

Parameters

- **name** (*str*) – not used
- **versions** (*str or list of str*) – a or the versions to condition on
- **version_column** (*str*) – version column name
- **time_column** (*str*) – time column name

Returns str – the condition for the versions

replace(*obj*)
Overwrite existing object without incrementing version

Parameters **obj** (RepoObject) – repo object to be overwritten

RepoObjectGitStorage

class RepoObjectGitStorage(*remote=None*, ***kwargs*)
Object storage with git support.

This storge stores all objects on disk in a local git storage. It provides functionality to push and pull from another git repo. Note that it handles all files in the same way as RepoObjectDiskStorage does (using methods from this storage).

commit(*message*, *force=True*)
Commits the changes

Parameters

- **message** (*str*) – Commit message
- **force** (*bool*) – If False, objecs will only be commited if integrity check succeeded.

Raises Exception – raises an exception if the integrity check fails

pull(*remote_name='origin'*)
Pull from the remote git repository

Parameters **remote_name** (*str*) – the name of the remote git repository. Defaults to 'origin'.

Raises

- Exception – raises an exception if the remote name is not available
- Exception – raises an error if the pull fails

push(*remote_name='origin'*)
pushes the changes to the remote git repository

Parameters **remote_name** (*str*) – name of the remote repository. Defaults to 'origin'.

Raises Exception – raises an exception if the remote does not exist

replace(*obj*)
Overwrite existing object without incrementing version

Parameters **obj** (RepoObject) – the repo object to be overwritten

NumpyHDFStorage

Module defining classes to store numpy data in hdf5 files.

This module provides implementations of the *pailab.ml_repo.repo_store.NumpyStore* using hdf5 file format.

class NumpyHDFRemoteStorage(*folder*, *remote_store=None*, *sync_get=False*, *sync_add=False*)

Storage working like NumpyHDFStorage locally but in addition provides synchronization with a remote.

This storage stores numpy data in hdf5 files in a directory. It works very similar to the *NumpyHDFStorage* with the difference that it synchronizes the data with a given remote (downloads and uploads the respective files).

Example

This example shows how to setup the storage so that the data is stored in a local directory and it can be synchronized ith googl ecloud storage:

```
>>> numpy = NumpyHDFRemoteStorage('C:\tmp\data')
>>> from pailab.ml_repo.remote_gcs import RemoteGCS
>>> remote = RemoteGCS(bucket='my_data')
>>> numpy.set_remote(remote)
```

Parameters

- **folder** (*str*) – folder where data is stored
- **remote_store** (*obj or dict*) – object representing a remote storage (e.g. `pailab.ml_repo.remote_gcs.RemoteGCS` for the google cloud storage) or dictionary defining the remote params so that it can be created
- **sync_get** (*bool*) – If True, tries to download data automatically if it does not exist locally, otherwise it checks only locally
- **sync_add** (*bool*) – If True, added data will be directly uploaded to the remote

add(*name*, *version*, *numpy_dict*)

Add numpy data from an object to the storage.

Parameters

- **name** (*str*) – the identifier of the object to add
- **version** (*str*) – the object version
- **numpy_dict** (*numpy dict*) – the numpy dictionary to add

get(*name*, *version*, *from_index=0*, *to_index=None*)

get the numpy object for a name and a version, rows can be used

Parameters

- **name** (*str*) – identifier of the object
- **version** (*str*) – version of the object
- **from_index** (*int*) – the index from which the data should be taken. Defaults to 0.
- **to_index** (*int or None*) – the index to which the data is returned (None means till the end). Defaults to None.

Raises

- `Exception` – raises an exception if no object with the name exists
- `Exception` – raises an exception if no object and with the version exists

Returns numpy array – the numpy object to return

pull()
Pull changes from an external repo

push()
Push changes to an external repo.

class NumpyHDFStorage(*folder*, *version_files=False*)
Storage using hdf5 files to store numpy data.

Example

Setup storage using folder `C:\temp\data`:

```
>>> store = NumpyHDFStorage('C:\temp\data')
```

Parameters

- **folder** (*str*) – main directory where the files will be stored
- **version_files** (*bool*) – If True, each version is contained in a separate file, otherwise all versions are in one file. If you like to work in a distributed environmnt (e.g. multiple users working in parallel) you should set this parameter to True so that no file merge is necessary. . Defaults to False.

add(***kw*)
Add numpy data from an object to the storage.

Parameters

- **name** (*str*) – the identifier of the object to add
- **version** (*str*) – the object version
- **numpy_dict** (*numpy dict*) – the numpy dictionary to add

append(***kw*)
append data to the an existing object

Parameters

- **name** (*str*) – the object identifier
- **version_old** (*str*) – the previous object version
- **version_new** (*str*) – the next object version
- **numpy_dict** (*numpy dict*) – the data to add as a numpy dictionary

get(***kw*)
get the numpy object for a name and a version, rows can be used

Parameters

- **name** (*str*) – identifier of the object
- **version** (*str*) – version of the object
- **from_index** (*int*) – the index from which the data should be taken. Defaults to 0.
- **to_index** (*int or None*) – the index to which the data is returned (None means till the end). Defaults to None.

Raises

- `Exception` – raises an exception if no object with the name exists
- `Exception` – raises an exception if no object and with the version exists

Returns numpy array – the numpy object to return

object_exists(*name*, *version*)
checks whether the object exists

Parameters

- **name** (*str*) – the identifier of the object
- **version** (*str*) – the version of the object

Returns bool – returns true if the object exists

trace(*aFunc*)
Trace entry, exit and exceptions.

1.4.4 tools

tools.tests

This module contains all tests.

class RegressionTest(*model*, *data*, *test_definition_version='last'*, *model_version='last'*, *data_version='last'*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Bases: *pailab.tools.tests.Test*

Regression test.

Note: In general, tests are automatically constructed and run using `pailab.ml_repo.repo.run_tests()`. As a user, there is nearly no need to construct a test by hand.

A regression test compares a specified measure of a reference model described by a label to the respective measure of the model to be tested. It fails, if the measure of the tested model is greater then a given tolerance of the reference measure, i.e. the test fails if

- measure-measure_ref < tol and an absolute tolerance is defined,
- measure-measure_ref < tol*measure_ref if a relative tolerance is used.

All the attributes specific for the regression test (i.e. not contained in the base class) are retrieved during the run of the test from the underlying testdefinition.

Parameters

- **model** (*str*) – Name of model for which the test is applied.
- **data** (*str*) – Name of dataset used in the test.
- **test_definition_version** (*str, optional*) – Defaults to latest version. Version of the tests's underlying *pailab.tools.tests.TestDefinition* that is used as basis for the test.
- **model_version** (*str, optional*) – Defaults to latest version. Version of the model the test is applied to.
- **data_version** (*str, optional*) – Defaults to latest version. Version of the data used in the test.

test_definition
Name of underlying *pailab.tools.tests.TestDefinition*.

Type str

model
Name of model for which the test is applied.

Type str

data
Name of dataset used in the test.

Type str

test_definition_version
Version of the tests's underlying *pailab.tools.tests.TestDefinition* that is used as basis for the test.

Type str

model_version
Version of the model the test is applied to.

Type str

data_version
Version of the data used in the test.

Type str

result
Describes the state of the test.

Type str, 'not run', 'failed', 'succeeded'

details
Contains details when test fails, otherwise empty dict.

Type dict

get_modifier_versions(*ml_repo*)
Get the modifier versions

Parameters **repo** (*MLrepository*) – repository used to get and store the data

Returns tuple of string, dict – return the object name and the modifiers

class RegressionTestDefinition(*reference='prod'*, *models=None*, *data=None*, *labels=None*, *measures=None*, *tol=0.001*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*, *relative=False*)
Bases: *pailab.tools.tests.TestDefinition*

Definition of a regression test.

A regression test compares a specified measure of a reference model described by a label to the respective measure of the model to be tested. It fails, if the measure of the tested model is greater then a given tolerance of the reference measure, i.e. the test fails if

- measure-measure_ref < tol and an absolute tolerance is defined,
- measure-measure_ref < tol*measure_ref if a relative tolerance is used.

Note: The tests needs the chosen measure(s) to be computed, therefore you have to take care that the measure has been added to the repo (using *pailab.ml_repo.repo.MLRepo.add_measure()*)

Examples

Add a test for the model *'my_model'* on a data set named *'test_data'* which checks if the maximum error of the model is not greater than 10% in relation to the error of the reference model defined by the label *'production_model'*

```
>>> test_def = RegressionTestDefinition(models=['my_model'], reference =
↪'production_model', data = ['test_data'], measures = ['max'], tol = 0.1,␣
↪relative = True)
>>> ml_repo.add(test_def)
```

Add a test applied to all models in the repo (always the latest versions of the models are used within the tests)

```
>>> test_def = RegressionTestDefinition(models=None, reference ='production_model
↪', data = ['test_data'], measures = ['max'], tol = 0.1, relative = True)
>>> ml_repo.add(test_def)
```

Parameters

- **models** (*iterable with str items, optional*) – Defaults to None. Iterable (e.g. list of str) returning names of the models to be tested.
- **data** (*iterable with str items, optional*) – Defaults to None. Iterable (e.g. list of str) returning names of the data used for testing.
- **labels** (*iterable with str items, optional*) – Defaults to []. Iterable returning labels defining models to be tested.
- **measures** (*[type], optional*) – Defaults to None. List of measures used in the test
- **reference** (*str, optional*) – Defaults to 'prod'. Label defining the reference model to which the measures are compared.
- **tol** (*float, optional*) – Defaults to 1e-3. Tolerance, if relative is False, the test fails if new_value-ref_value < tol, otherwise if new_value-ref_value < tol*ref_value.
- **relative** (*bool, optional*) – Defaults to False.
- **repo_info** (RepoInfo, *optional) Defaults to RepoInfo()* –

models
List of strings defining the models to be tested.

Type list of str

labels
List of strings defining the labels to be tested.

Type list of str

data
List of strings defining the names of the data to be tested

Type list of str

class Test(*model*, *data*, *test_definition_version='last'*, *model_version='last'*, *data_version='last'*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)

Bases: `pailab.ml_repo.repo.Job`

Base class for all tests.

Note: In general, tests are automatically constructed and run using `pailab.ml_repo.repo.run_tests()`. As a user, there is nearly no need to construct a test by hand.

Parameters

- **model** (*str*) – Name of model for which the test is applied.
- **data** (*str*) – Name of dataset used in the test.
- **test_definition_version** (*str, optional*) – Defaults to latest version. Version of the tests's underlying *pailab.tools.tests.TestDefinition* that is used as basis for the test.
- **model_version** (*str, optional*) – Defaults to latest version. Version of the model the test is applied to.
- **data_version** (*str, optional*) – Defaults to latest version. Version of the data used in the test.
- **model** – Name of model for which the test is applied.
- **data** – Name of dataset used in the test.
- **test_definition_version** – Defaults to latest version. Version of the tests's underlying *pailab.tools.tests.TestDefinition* that is used as basis for the test.
- **model_version** – Defaults to latest version. Version of the model the test is applied to.
- **data_version** – Defaults to latest version. Version of the data used in the test.

test_definition
Name of underlying *pailab.tools.tests.TestDefinition*.

Type str

model
Name of model for which the test is applied.

Type str

data
Name of dataset used in the test.

Type str

test_definition_version
Version of the tests's underlying *pailab.tools.tests.TestDefinition* that is used as basis for the test.

Type str

model_version
Version of the model the test is applied to.

Type str

data_version
Version of the data used in the test.

Type str

result
Describes the state of the test.

Type str, 'not run', 'failed', 'succeeded'

details
Contains details when test fails, otherwise empty dict.

Type dict

class TestDefinition(*models=None*, *data=None*, *labels=[]*, *repo_info=<pailab.ml_repo.repo_objects.RepoInfo object>*)
Bases: *pailab.ml_repo.repo_objects.RepoObject*, abc.ABC

Abstract base class for all test definitions.

A test definition defines the framework such as models and data the tests are applied to. It also provides a create method which creates the test cases for a special model and data version.

Parameters

- **models** (*iterable with str items, optional*) – Defaults to None. Iterable (e.g. list of str) returning names of the models to be tested.
- **data** (*iterable with str items, optional*) – Defaults to None. Iterable (e.g. list of str) returning names of the data used for testing.
- **labels** (*iterable with str items, optional*) – Defaults to []. Iterable returning labels defining models to be tested.
- **repo_info** (RepoInfo, *optional) Defaults to RepoInfo()* –

models
List of strings defining the models to be tested.

Type list of str

labels
List of strings defining the labels to be tested.

Type list of str

data
List of strings defining the names of the data to be tested

Type list of str

create(*ml_repo: pailab.ml_repo.repo.MLRepo*)
Create a set of tests for models of the repository.

Parameters

- **ml_repo** (MLRepo) – ml repo
- **models** (*dict, optional*) – Defaults to {}. Dictionary of model names to version numbers to apply tests for. If empty, all latest models are used.
- **data** (*dict, optional*) – Defaults to {}. Dictionary of data the tests are applied to. If empty, all latest test- and train data will be used.
- **labels** (*list, optional*) – Defaults to []. List of labels to which the tests are applied.

Returns [description]

Return type [type]

tools.tree

This module contains all functions and classes for the MLTree. The MLTree buils a tree-like structure of the objects in a given repository. This allows the user to access objects in a comfortable way allowing for autocompletion (i.e. in Jupyter notebooks).

To use it one can simply call the *pailab.tools.tree.MLTree.add_tree()* method to add such a tree to the current repository:

```
>>from pailab.tools.tree import MLTree
>>MLTree.add_tree(ml_repo)
```

After the tree has been added, one can simply use the tree. Here, using autocompletion makes the basic work wih repo objects quite simply. Each tree node provides useful functions that can be applied:

- `load` loads the object of the given tree node or the child tree nodes of the current node. a After calling load the respective nodes have a new attribute `obj` that contains the respective loaded object. To load all objects belonging to the models subtree like parameters, evaluations or measures one can call:

```
>> ml_repo.tree.models.load()
```

- `history` lists the history of all objects of the respective subtree, where history excepts certain parameters such as a range of versions or which repo object information to include. To list th history of all training data just use:

```
>> ml_repo.tree.training_data.history()
```

- `modifications` lists all objects of the respective subtree that have been modified and no yet been committed.

There are also node dependent function (depending on what object the node represents).

class MLTree(*ml_repo*)

Bases: `object`

static add_tree(*ml_repo*)

Adds an MLTree to a repository.

Parameters **ml_repo** (MLRepo) – the repository the tre is added

modifications()

Return a dictionary of all objects that were modified but no yet commited to the repository.

Returns dictionary mapping object ids to dictionary of the modified attributes

Return type dict

reload(***kwargs*)

Method to reload the tree after objects have been added or deleted from the repository.

tools.interpretation

This module contains functions for model agnostic interpretation methods.

class ICE_Results

Bases: `object`

compute_cluster_average(*ice_results_2*)
[summary]

Parameters **ice_results_2** (*[type]*) – [description]

Returns Matrix containing the average of values from ice_results_2 over the different clusters from this result.

Return type numpy matrix

compute_ice(*ml_repo*, *x_values*, *data*, *model=None*, *model_label=None*, *model_version='last'*, *data_version='last'*, *y_coordinate=0*, *x_coordinate=0*, *start_index=0*, *end_index=-1*, *cache=False*, *clustering_param=None*, *scale=''*)
Compute individual conditional expectation (ice) for a given dataset and model

Parameters

- **ml_repo** (*MLRepo*) – MLRepo used to retrieve model and data and be used in caching.
- **x_values** (*list*) – List of x values for the ICE.
- **data** (*str, DataSet, RawData*) – Either name of data or directly the data object which is used as basis for ICE (an ICE is computed at each datapoint of the data).
- **model** (*str, optional*) – Name of model in the MLRepo for which the ICE will be computed. If None, model_label must be specified, defining the model to be used. Defaults to None.
- **model_label** (*str, optional*) – Label defining the model to be used. Defaults to None.
- **model_version** (*str, optional*) – Version of model to be used for ICE. Only needed if model is specified. Defaults to RepoStore.LAST_VERSION.
- **data_version** (*str, optional*) – Version of data used. Defaults to RepoStore.LAST_VERSION.
- **y_coordinate** (*int or str, optional*) – Defines y-coordinate (either by name or coordinate index) for which the ICE is computed. Defaults to 0.
- **x_coordinate** (*int or str, optional*) – Defines x-coordinate (either by name or coordinate index) for which the ICE is computed. Defaults to 0.
- **start_index** (*int, optional*) – Defines the start index of the data to be used in ICE computation (data[start_index:end_index] will be used). Defaults to 0.
- **end_index** (*int, optional*) – Defines the end index of the data to be used in ICE computation (data[start_index:end_index] will be used). Defaults to -1.
- **cache** (*bool, optional*) – If True, results will be cached. Defaults to False.
- **clustering_param** (*dict or None, optional*) – Dictionary of parameters for method functional_clustering that is called if the parameter is not None and applies functional clustering to the ICE curves.
- **scale** (*str or int, optional*) – String defining the scaling for the functions before functional clustering is applied. Scaling is perfomred by dividing the vector of the y-values of the ICE by the respective vector norm defined by scaling. The scaling must be one of numpy's valid strings for linalg.norm's ord parameter. If string is empty, no scaling will be applied. Defaults to ''.

Returns result object containing all relevant data (including functional clustering)

Return type *ICE_Results*

functional_clustering(*x*, *scale=''*, *n_clusters=20*, *random_state=42*)

Given a set of different 1D functions (represented in matrix form where each row contains the function values on a given g
method clusters those functions and tries to find typical function structures.

Parameters

- **x** (*numpy matrix*) – Matrix containing in each row the function values at a datapoint.
- **n_clusters** (*int, optional*) – Number of clusters for the functional clustering of the ICE curves. Defaults to 0.
- **random_state** (*int, optional*) – [description]. Defaults to 42.
- **scale** (*str or int, optional*) – String defining the scaling for the functions before functional clustering is applied. Scaling is perfomred by dividing the vector of the y-values of the ICE by the respective vector norm defined by scaling. The scaling must be one of numpy's valid strings for linalg.norm's ord parameter. If string is empty, no scaling will be applied. Defaults to ''.

Returns Vector where each value defines the corresponding cluster of the respective function (x[i] is the cluster the i-th function belongs to). numpy matrix: Contains in each row th distance to each cluster for the respective function. numpy matrix: Contains in each row a cluster centroid.

Return type numpy vector

generate_prototypes(*ml_repo*, *data*, *n_prototypes*, *n_criticisms*, *data_version='last'*, *use_x=True*, *data_start_index=0*, *data_end_index=-1*, *metric='rbf'*, *witness_penalty=1.0*, ***kwds*)

This methods computes for a given test/training dataset prototypes and criticisms and adds them as separate test data sets to the repository.

This methods computes for given test/training dataset prototypes and criticisms, i.e. datapoints from th given set that are typical representatives (prototypes) and datapoints that are not well representatives (criticisms). Here, a simple greedy algorithm using MDM2 is used to compute the prototypes and a witness function together with some simple penalty are used to compute the criticisms (see e.g. C. Molnar, Interpretable Machine Learning).

Parameters

- **ml_repo** (MLRepo) – The repository used to retrieve data and store prototypes/criticisms.
- **data** (*str*) – Name of data used for computation.
- **n_prototypes** (*int*) – Number of prototypes.
- **n_criticisms** (*int*) – Number of criticisms.
- **data_version** (*str*) – Version of data to be used. Defaults to RepoStore.LAST_VERSION.
- **use_x** (*bool*) – Flags that determine if prototypes are computed w.r.t. x or y coordinates. Defaults to True.
- **data_start_index** (*int*) – Startindex of data used.
- **data_end_index** (*int*) – Endindex of data used.
- **metric** (*str or callable, optional*) – The metric to use when calculating kernel between instances in a feature array. If metric is a string, it must be one of the metrics in sklearn.metrics.pairwise.PAIRWISE_KERNEL_FUNCTIONS. If metric is precomputed, X is assumed to be a kernel matrix. Alternatively, if metric is a callable function, it is called on each pair of instances (rows) and the resulting value recorded. The callable should take two arrays from X as input and return a value indicating the distance between them. Currently,

sklearn provides the following strings: ‘additive_chi2’, ‘chi2’, ‘linear’, ‘poly’, ‘polynomial’, ‘rbf’,

‘laplacian’, ‘sigmoid’, ‘cosine’

- **witness_penalty** (*float*) – Penalty parameter to include some penalty to avoid to close criticisms.
- ****kwds** – optional keyword parameters Any further parameters are passed directly to the kernel function.

Raises Exception – If sklearn is not installed

Returns List of indices defining the datapoints which are the resulting prototypes. list of int: List of indices defining the datapoints which are the resulting criticisms.

Return type list of int

1.4.5 job runner

class JobRunnerBase

Bases: abc.ABC

Baseclass for all job runners so that they can be used together with the MLRepo

add(*job_name*, *job_version*, *user*)
[summary]

get_info(*job_name*, *job_version*)
[summary]

class JobState

Bases: enum.Enum

Job states

class SQLiteJobRunner(*sqlite_db_name*, *repo*, *sleep=1*, *steps_to_heartbeat=30*)

Bases: *pailab.job_runner.job_runner.JobRunnerBase*

add(*job_name*, *job_version*, *user*)
[summary]

close_connection()
Closes the database connection

get_info(*job_name*, *job_version*)
[summary]

get_waiting_jobs()
Return list of open jobs

Returns list containing tuples of job names and versions of the jobs currently waiting

Return type list of tuples

class SimpleJobRunner(*repo*, *throw_job_error=False*)

Bases: *pailab.job_runner.job_runner.JobRunnerBase*

add(*job_name*, *job_version*, *user*)
[summary]

get_info(*job_name*, *job_version*)
[summary]

get_waiting_jobs()
Return list of open jobs

Returns empty list because by construction, this JobRunner can only return something if the jobs have been finished

1.4.6 externals

pailab.externals.sklearn_interface

Module for pailab to sklearn

This module defines all necessary objects and functions to use sklearn from within pailab.

class SKLearnModel(**args*, ***kwargs*)
Class to store all sklearn models in pailab's MLRepo

class SKLearnModelParam(**args*, ***kwargs*)
Interfaces the parameters of the sklearn algorithms

class SKLearnPreprocessingParam(**args*, ***kwargs*)
Interfaces the parameters of the sklearn algorithms

class SKLearnPreprocessor(**args*, ***kwargs*)
Class to store all sklearn preprocessor

add_model(*repo*, *skl_learner*, *model_name=None*, *model_param=None*, *preprocessors=None*)
Adds a new sklearn model to a pailab MLRepo

Parameters

- **repo** (*[type]*) – [description]
- **skl_learner** (*[type]*) – [description]
- **model_name** (*[type], optional*) – Defaults to None. [description]
- **model_param** (*[type], optional*) – Defaults to None. [description]
- **preprocessors** (*list of strings, optional*) – List of used preprocessors

add_preprocessor(*repo*, *skl_preprocessor*, *preprocessor_name=None*, *preprocessor_param=None*)
Adds a new sklearn preprocessor to a pailab MLRepo

Parameters

- **repo** (*[type]*) – [description]
- **preprocessor** (*Class*) – The sklearn preprocessor class
- **preprocessor_name** (*[type], optional*) – Defaults to None. [description]
- **preprocessor_param** (*[type], optional*) – Defaults to None. [description]

eval_sklearn(*model*, *data*)
Function to evaluate an sklearn model

Parameters

- **model** (*[type]*) – [description]
- **data** (*[type]*) – [description]

Returns [description]

Return type [type]

pailab.externals.tensorflow_keras_interface

add_model(*repo*, *tensorflow_keras_model*, *model_name*, *loss*, *epochs*, *batch_size*, *optimizer='ADAM'*, *optimizer_param={}*, *tensorboard_log_dir=None*, *validation_split=0.0*)

Adds a new tensorflow-keras model to a pailab MLRepo

:param : param repo (MLRepo): ml repo :param : param tensorflow_keras_model (keras model): the model created with tensorflows keras (not yet compiled) :param : param model_name (str): name of model used in repo :param : param loss (str): lossfunction :param : param epochs (int): number of epochs used :param : param batch_size (int): batch size

eval_keras_tensorflow(*model*, *data*)

Function to evaluate a keras-tensorflowmodel

Parameters

- **model** (*TensorflowKerasModel*) – model to evaluate
- **data** (DataSet) – dataset on which model is evaluated

Returns evaluated data

Return type numpy-data

pailab.externals.pytorch_interface

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

O

P

R

S

T

V