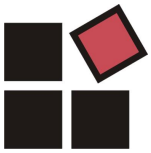
AG Technische Informatik
Institut für Informatik
Universität Osnabrück



Smart Embedded Systems Group
Department of Computer Science
University of Osnabrück

# Tutorial:

# Modular Python UOSLib

## A framework for evaluating and developing on-line incremental *machine learning* algorithms

## Contents

# 1 Introduction

## 1.1 PyUOSLib

The PyUOSLib is a framework for evaluating and developing incremental on-line learning algorithms in python. The main goals of this framework are to make scientific experiments reproducible, to make experiment descriptions and code exchangeable and to support a data flow perspective on data stream experiments. To this end the PyUOSLib separates the code and experiment description. The code is organized in modules written in python, while the experiment description is done in a .json-file explicitly specifying the complete module structure, parametrization, order of execution and data flow. But before we get to the details let's have a look at the incremental learning problem and we will meet the basic features of the PyUOSLib on the way.

## 1.2 Incremental Learning Problem

The basic incremental learning problem is easily described as a game played in rounds. Within each round first an instance $x \in \mathbb{R}^n$ is created in order to query a learner. This instance may be draw according to a random distribution or may be generated by a deterministic process. Second, based on the instance $x$ the learner is asked to predict the outcome $\hat{y} = p(x)$ of a certain event, e.g. measuring the position of a pendulum. Third, the actual outcome of the event $y = f(x)$ is observed making the learner suffer a loss $l(p(x), y)$ due to a possible deviation between the actual outcome $y$ and the learner's prediction $p(x)$. Finally, the instance $x$ is labelled by the actual outcome $y$ forming the learning date $(x, y)$ which is presented to the learner and allows it to adjust its internal mapping $p$ for predictions in order to minimize the accumulated loss it will suffer on future instances. From the learner's perspective in each round it receives an instance $x$, predicts $p(x)$, suffers some loss $l(p(x), y)$ and gets the learning date $(x, y)$. Through this iterative procedure the learner adapts its internal mapping $p$ for prediction in order to fit some unknown target function $f$. If the learner is able to perfectly fit the unknown target function $f$ represented by the series of learning data $(x, y)$, it will be able to predict the outcome for each instance just as well, but this is usually not the case.
The crucial part in the procedure described above is updating the internal mapping $p$ of the learner, which is used to perform predictions. The literature on incremental on-line learning provides a rich variety of approaches that tackle the incremental learning problem and allow the learner to minimize its expected cumulative loss, but this tutorial will not cover it. Some algorithms present in the PyUOSLib will be used in this tutorial without explaining their details as this tutorial focuses on how to use the PyUOSLib.
From an experimenters point of view the generation of instances and their labelling is of major concern as they define the frame of the evaluation. There are different challenges a learner may face in an incremental learning task like noise, outliers, target drift and shift, sparse and non-i.i.d. learning data, non-linearity, non-monotonicity and other aspects concerning the properties of the unknown target function or the learning data. The PyUOSLib provides the framework to let a learner face different

challenges in a controlled and reproducible way and thus enables an experimenter to evaluate different aspects of different learners and compare them in a uniform way. The important abstraction we use throughout this tutorial, and the PyUOSLib in general, is the *incremental learning system* bundling all the tasks the learner in the incremental learning problem needs to perform. This abstraction is useful within the data flow orientated perspective of the PyUOSLib as it allows us to treat the learner, i.e. the incremental learning system, as one sound module. This module provides an interface to its environment that encapsulates the prediction $p(x)$ of a given instance $x$ and the adjustment of the internal mapping $p$ of the learner in one call, thus taking care of second and last step of the incremental learning problem. The incremental learning system is given a learning date $(x, y)$ and first does its prediction for the instance $x$ followed by its internal adjustment based on the given label $y$. While this procedure may let the data snooping of the given label $y$ appears reasonable and in fact doable, the main reason for this abstraction is to save computing time. In a more general incremental learning problem the learner may be asked to predict some instance $z$ and is given a learning date $(x, y)$ with the instances $x$ and $z$ being different.
The incremental learning system abstraction explicitly denies the calculation of the loss $l(p(x), y)$ as there are many different ways to evaluate the behaviour of a learner and the learner itself does not need to know about it. In fact, when evaluating different learning algorithms we usually look for more sophisticated ways of estimating the performance of the learning system at hand than just tracking its prediction error. Other directions for the performance evaluation are to look at the overall prediction mapping, to track the internal parameters of the learning system or to measure the performance indirectly by measuring the performance of a related task. All of this is possible in the PyUOSLib and is part of designing incremental learning experiments. Nevertheless, most of the time it is sufficient to look at the cumulative loss when comparing different learning algorithms or parameter settings, as it is a single value representing the overall performance of a complete experiment.

## 1.3 Basic principles of the PyUOSLib

The PyUOSLib is a framework for making experiments based on the idea of the incremental learning problem. The guiding principles for the whole framework are listed below:

- Modular design:
  - One module, one task, one type of output
  - Each module may specify its own input interface
  - A data source has no input
  - A data sink has no output

- Data flow orientation:
  - The experiment description is based on defining the data flow for the experiment
  - The experiment description contains all module descriptions
  - Each module description precisely determines the behaviour of the described module
  - Each module as a unique name
  - Module connections are defined by referring to the names of the input modules
  - Modules are executed in the order of their definition in the experiment description
  - References to subsequent modules in the execution list yield the module's output of the previous stage

## 1.4 Module representation in python

This description about the module representation of a PyUOSLib module in python is only loosely connected to the following step by step tutorial as it provides insights to the mechanics of the PyUOSLib rather than how to use them. The overview here is fundamental for developing PyUOSLib modules, but is of marginal use for designing and running experiments with the PyUOSLib.
Every module in the PyUOSLib is represented as a python class that inherits the `BasicProcessingModule`. This fundamental python class provides the interface for handling different modules in one experiment in a uniform way. The interface provides functions to define the user-defined behaviour of the module as well as framework specific functions that must not be changed. The user-defined functions are called at specific stages of the experiment evaluation and thus allow to control the behaviour of the module at these points. The list of user-defined functions and their description is given below:

- `__init__(foot)`

  When initialising a module, you must call the constructor of the super class:

  ```
  BasicProcessingModule.__init__(self, foot)
  ```

  Further instructions are user-specific.

- `prepare(self, antecessor)` (optional)

  This function is call after initialising all modules of the module list defining the experiment and before executing the first step of the experiment. The interface of this function is framework specific and fixed. The argument antecessor contains a dictionary of references to all input modules. Using this function is optional when defining modules. All instructions are fully user-specific.

- `__call__(self, argIn, index)`

  This function is called once in every step of an experiment when the module is asked to update its output. The behaviour of this function fundamentally determines the task of the module and is fully user-defined. The definition of the argument list is mostly user-specific as the only input argument that is always given by the framework is the index.

- `end()` (optional)

  This function is called after executing every step of the experiment giving room for finishing instructions and is fully user-defined.

- `reset()` (optional)

  This function is called before rerunning an experiment in order to re-establish the initial state of the module and thus making it ready to run the experiment again, potentially with different parameters. The behaviour here is fully user-specific be must fit the actions in `__init__(foot)` and `prepare(self, antecessor)` for a proper reinitialisation.

The framework specific functions in `BasicProcessingModule` are:

- `connectInputs(self, names)`

  This function is called after initialising all modules and connects their input, thus essentially forming the data flow of the experiment.

- `run(index)`

  This function gathers all output of the modules defined as inputs the a module and handles them as input argument to the `__call__` function of the module together with the current `index`

In the first steps section some module implementations and thus different implementations of the user-specific functions will be presented. As marked in the above list not all functions need to be implemented in order to fully specify a module's behaviour, but at least the `__init__` and the `__call__` function need to be there. The default for the optional functions is to do nothing.

**Executing an experiment** The procedure for executing an experiment follows the basic incremental learning problem with some necessary extensions for initialising the whole setting:

1. Read the experiment description from a .json file
2. Initialise all modules defined in the experiment description
3. Connect all modules according to the data flow in the experiment description

After this initial stage, the actual evaluation of the experiment is performed:

1. In each simulation step all modules are called with their respective input arguments in the order of their definition until the given number of steps is reached.

2. When all simulation steps are processed the `end()` function of each module is called in the order of their definition.

After executing the experiment one can call the `reset()` function in order to prepare the experiment for an additional run.

# 2 First Steps

## 2.1 Step 1: A simple learning experiment

In this section we will develop the description of a simple learning experiment in a .json file for the PyUOSLib. Therefor, we look at the data flow of the basic incremental learning problem and how we can map it to a module structure and module connections that fit to the incremental learning system abstraction. Figure 1 shows this data flow together with a plotting module gathering all the performance measures.
The general structure in figure 1 needs to be filled with certain elements for each module in order to frame a particular experiment. The PyUOSLib uses the experiment as the fundamental frame for any assignment. Within this frame all module descriptions and their connections are placed. At the experiment level general aspects like the number of learning steps and the random seed for the random number generator are set and the experimenter has the opportunity to declare global parameters. The most extensive part of the experiment description is the list of modules as each module description contains all parameters relevant for this module and specifies the connection to other modules. The description of the module connections is source orientated as every module in the PyUOSLib encapsulates a single task each module produces only one kind of output, thus making it sufficient for each module to name the modules it receives data from. The execution order of the modules is given by the order of their definition in the module list of the experiment. This way inputs from modules at a later execution stage will provide their output from the last iteration or their initial output.

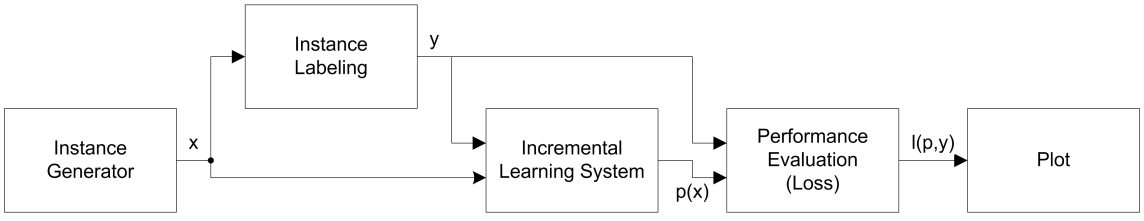



Figure 1: Data flow of the basic incremental learning problem equipped with a plotting module as a data sink in order to visualize the development of the performance measure.

### 2.1.1 The experiment description

Equipped with the background on PyUOSLib experiments we jump to the experiment description of the simple sine learning task and see what the PyUOSLib does with a .json file experiment description. Next will take a closer look at the experiment description and map the modules in figure 1 to the elements of the experiment description one by one. The .json experiment description is structured as a dictionary containing the list of modules as an item. The full experiment description and its connection to the data flow in figure 1 are shown in figure 2.
As highlighted by the colouring in figure 2, each module of the data flow in figure 1 is represented as one element in the list of modules within the experiment description. Each single module description starts by naming the module with an `ID` tag. This `ID` must be unique for the module within the entire experiment as other modules will refer to it using its ID. The next important and mandatory element of each module description is the `module` tag defining the type of the module, i.e. the function this module performs. As there are many different possible functions a module can implement and thus a potentially large number of the modules, the modules can be organized in `collection`. A `collection` is a single python file containing different module implementations. The collections are meant to group similar modules that belong to a certain topic or family of functions. The `collection` tag indicates the membership of a module to the named collection and refers to the name of the file where the module implementation can be found. If a module does not belong to a collection the filename containing the implementation must be the exact module type description.
All files containing module implementations can be further organized in different folders in order to group different functions to certain projects or wider topics. The `path` tag tells the PyUOSLib where to look for a module or collection and can only direct to subfolders of the `Modules` folder and subsequent subfolders. Paths to nested subfolders are defined using a '.' (period, full stop) as the path separator. All other tags of the module description are optional or module specific.
The most relevant optional tag in a module description is the `input` tag as it defines the data flow through the module. The `input` tag is optional as there are modules that only generate data like the instance generator in our simple sine example. The inputs of a module are described as a dictionary with each item in the dictionary representing one input to the module. The keys of the items represent the internal name for the referenced value and are usually fixed for a particular module. The values of the items contain the name of the module each input is fed from. This way, each module can define its own interface and rely on the correct renaming of its inputs for a unified internal usage.
To sum up, the most important tags in each module description are: `ID`, `path`, `collection`, `module` and `input`. Many modules performing simple tasks are fully described using only these tags, but usually there are some additional module specific tags like the `legend` in the `Plot` module. Whether these module specific tags are optional or not depends on the implementation of the module.
Before we start running the experiment we take a brief look at each module defined in our experiment description. The instance generator is realized using a uniform distribution as we want to learn the sine based on uniformly distributed learning

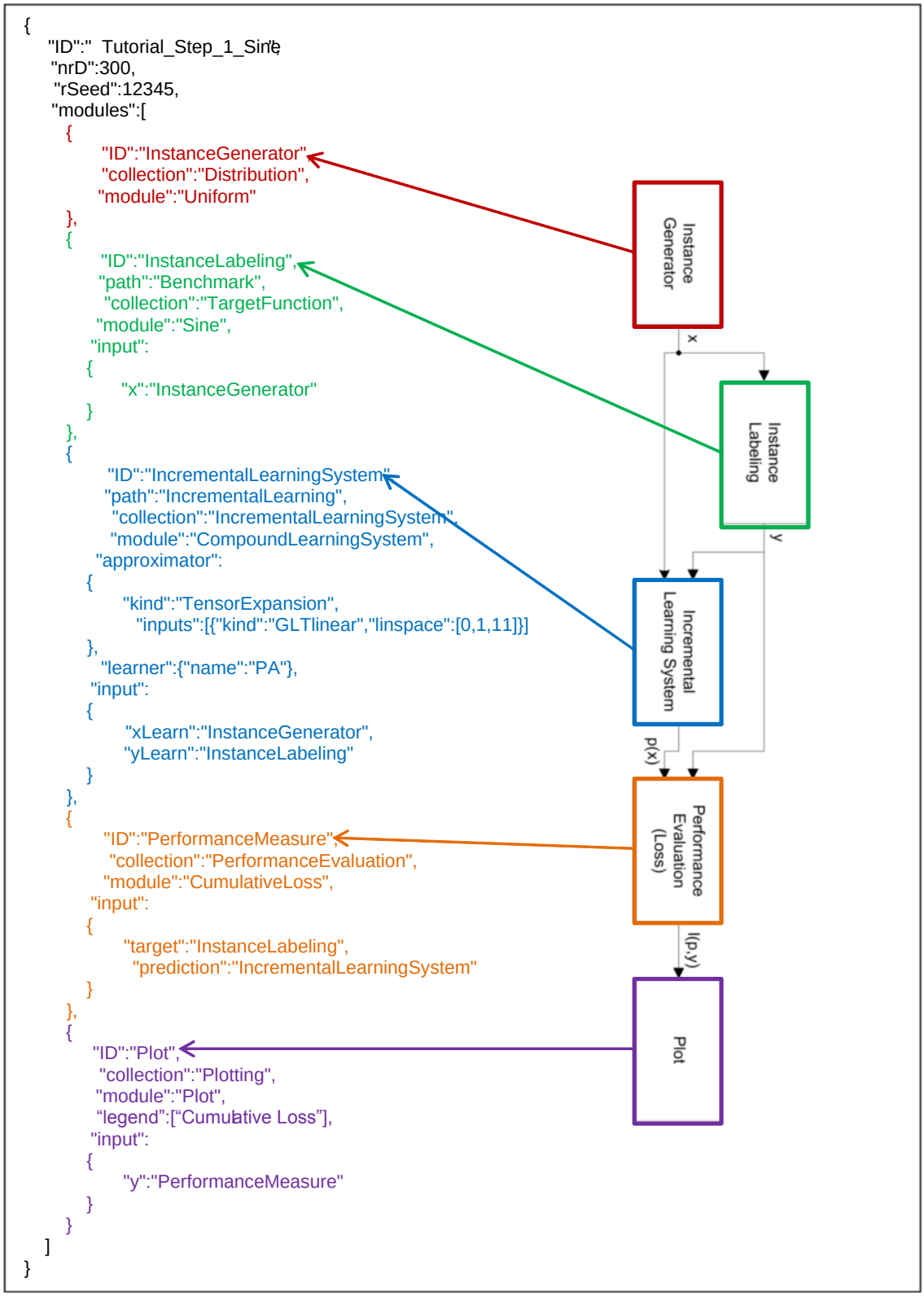
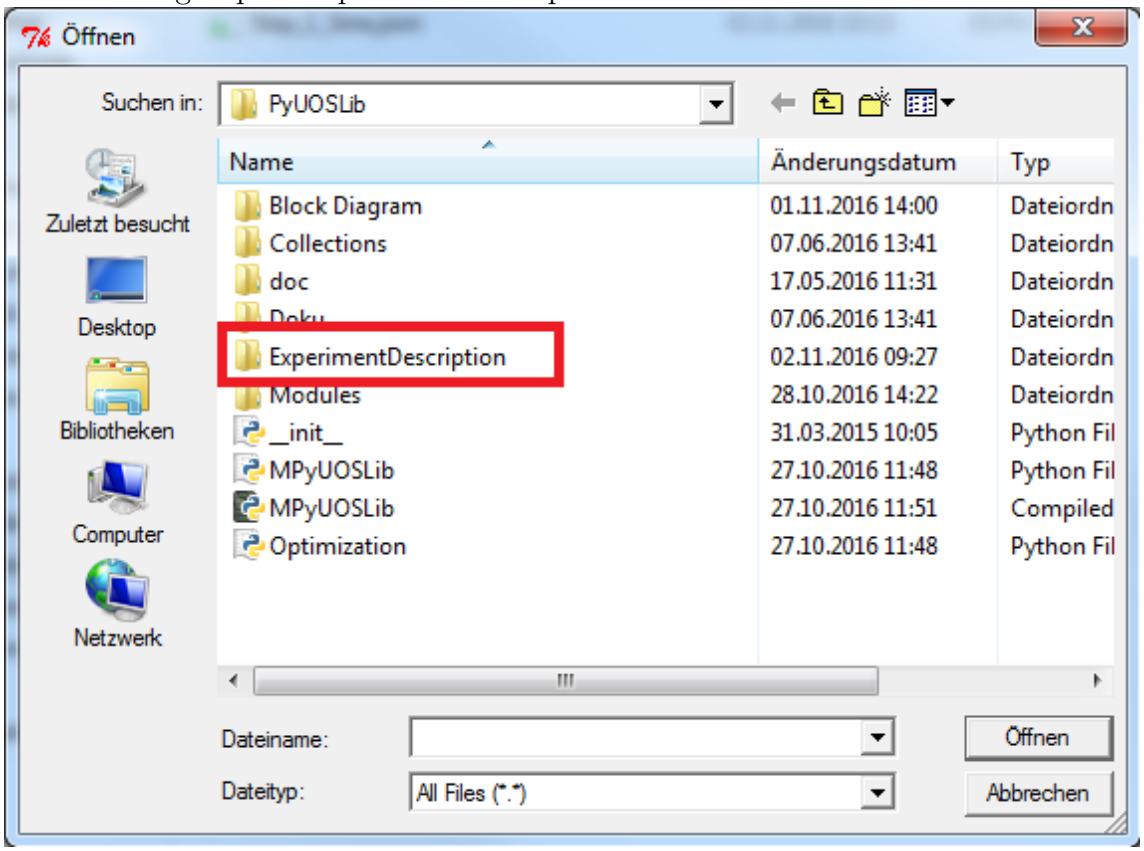



Figure 2: Matching of the basic incremental learning problem data flow and its PyUOSLib representation as a .json file experiment description.

data. Being a generator module, it has no input. The next module in the experiment description performs the instance labelling using the target function sine. This module receives input from the instance generator and outputs the corresponding label. The incremental learning system module performs the learning in the way described in the introduction of the related abstraction. We will not look at the details of the learning here and just observe that this module receives input from two different modules, the instance generator and the instance labelling. The output of the incremental learning module is the prediction based on the given instance before incorporation the learning date. In order to complete the basic incremental learning problem the next module measures the performance of the learning system by tracking its cumulative loss and, as the loss is calculated based on the difference between the prediction of the learning system and the actual label, this module receives input from the incremental learning system and the instance labelling. Finally, the plot module gathers all the performance measure over time and plots the development of the cumulative loss.

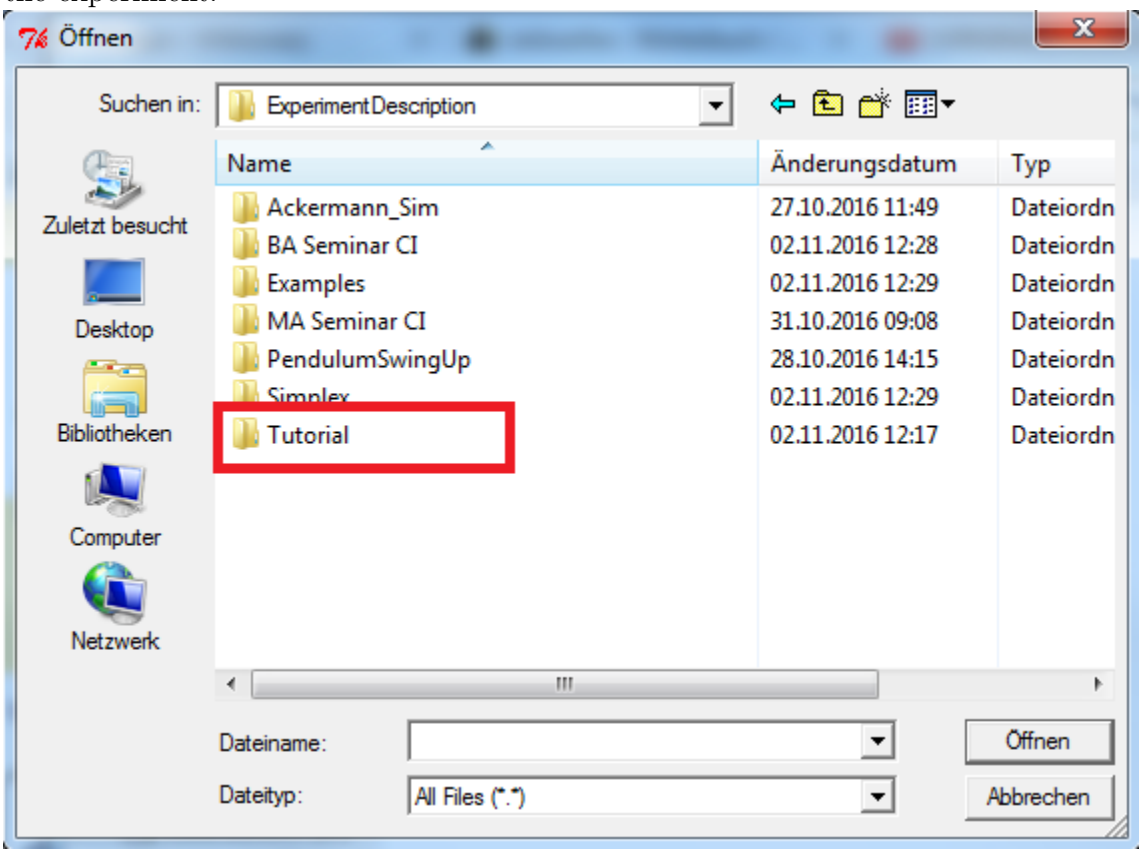
### 2.1.2 Running the experiment

Running an experiment in the PyUOSLib starts by executing `MPyUOSLib.py` without any additional parameters[1]. At first you are asked to choose a file containing an experiment description. The folder `ExperimentDescription` and its subfolders contain and group all experiment descriptions:

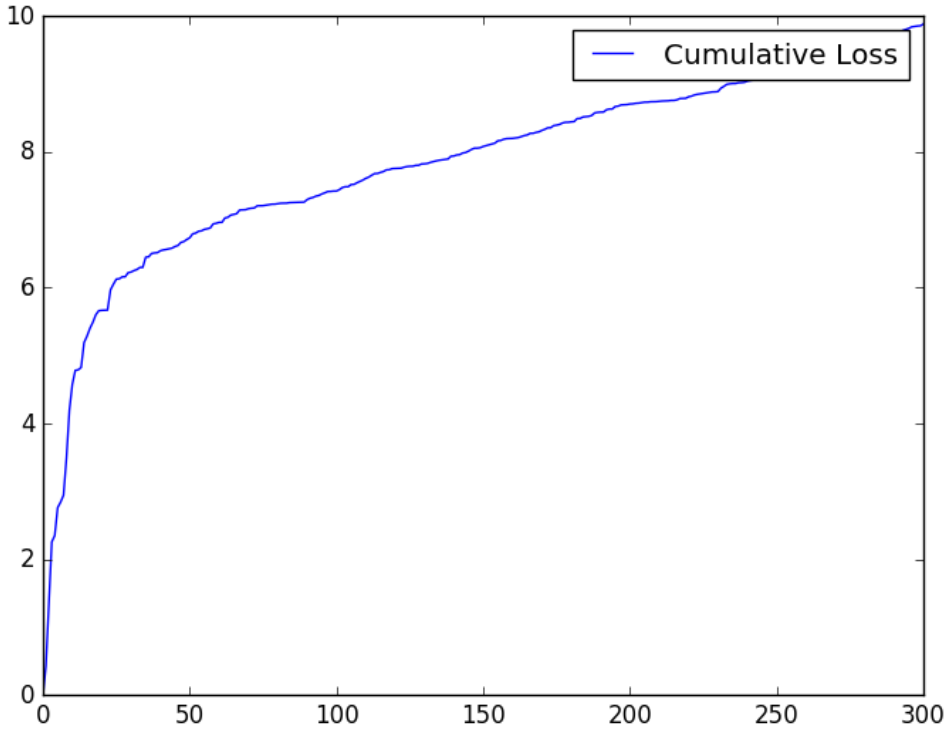



[1] Alternatively you can run the *MPyUOSLib.py* with an *–file* and name the file you want to execute with the relative path to *PyUOSLib* include the *ExperimentDescription*

Here we pick the folder `Tutorial` and select `Step_1_Sine.json` in order to perform the experiment.




The resulting plot should look like this:

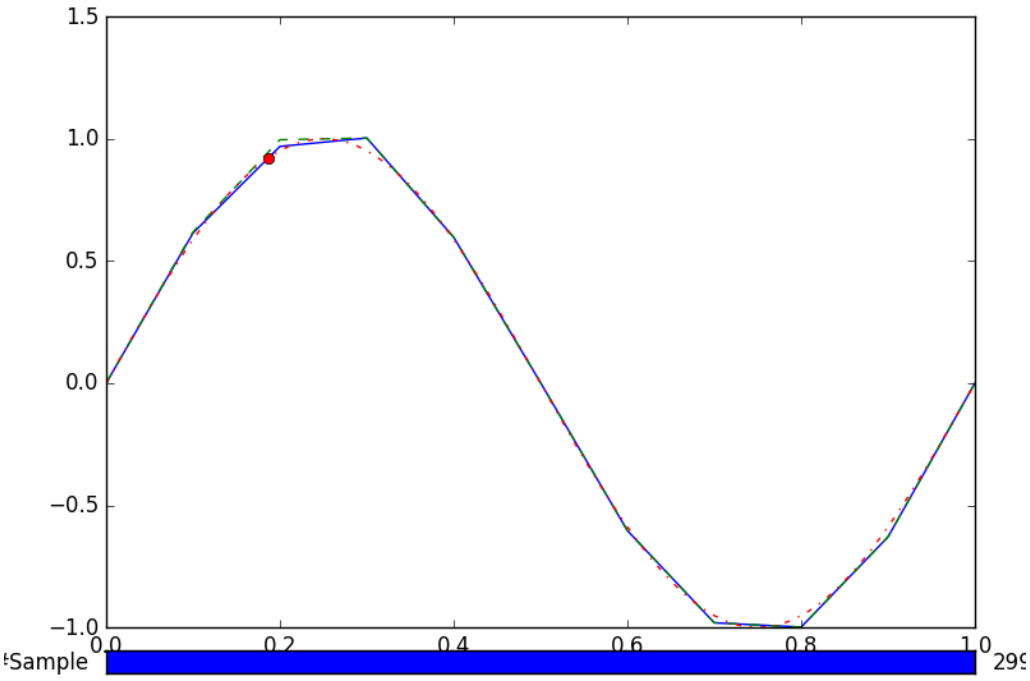



It shows the development of the cumulative loss over the 300 steps of the learning process. Throughout the first 30 learning steps the cumulative loss rises to a value of about 6 at a high rate. Between learning step 30 and 50 the slope flattens. From learning step 50 on, the slope of the cumulative loss is nearly constant and low compared to the initial one. This usually indicates a convergence of the learning process in some sense, but from this plot we cannot tell if we successfully learned the sine function. In order to get a deeper understanding of the learning process, we need to take a closer look at the internal function of the learning and the learning data in every step. Therefor, we extend our experiment description and include a module for tracking the entire learning history. To do so, we save a copy of the `Step_1_Sine.json` file named as `Step_1_Sine_History.json` and add the following module description to the module list and remember to separate it from the previous one by a `,` as the list separator:

```
{
    "ID":"LearningHistory",
    "collection":"Plotting",
    "module":"PlotLearningHistory",
    "samplingDomain":[0,1],
    "samplingSize":101,
    "legend":"Training Data",
    "input":
    {
        "xLearn":"InstanceGenerator",
        "yLearn":"InstanceLabeling",
```

```
        "ILS":"IncrementalLearningSystem",
        "GroundTruth":"InstanceLabeling"
    }
}
```

Now run the experiment `Tutorial_Step_1_Sine_History.json`. The successful pass of the experiment should output two plots: The cumulative loss plot we already know and a plot showing the history of the learning process which at first should look like this:




This additional plot shows a marker and three different lines and has a slider at the bottom for interaction. The marker (red circle) shows the learning date $(x, y)$ of the current learning step. The three lines show the target function (red, dash dotted line), the approximation before incorporating the learning date (green, dashed line) and the approximation after incorporating the learning date (blue solid line). The slider at the bottom allows selecting the learning step plotted in the axes. Thus we can follow the impact of every single learning date on the approximation by sliding through the different learning steps. The above picture shows the result of the learning after 300 steps and reveals that the approximation indeed converged to the sine given the limitations of the used approximation, here a first order b-spline on eleven equally spaced nodes.

### 2.1.3 Global parameters

If we take a closer look at the `PlotLearningHistory` we observe an annoying fact. We need to explicitly define the domain where we want to sample the approximation

and the target function possibly, too. In some special cases we might want to domain of the learning data and thus the domain of the instance generator to differ from the sampling domain, e.g. if we are only interested in the behaviour in a small region, but in general we want the domains to be equal and thus we want the two module specific parameters to refer to the same value. The PyUOSLib allows the use of global parameters in a restricted way. The parameters are defined at experiment level using a dictionary that maps their names to the given values. There are two ways of accessing these parameters. First, each module has a dictionary containing these exact values as a member variable. Second, during the `BasicProcessingModule` initialisation of each module the values of each tag given in the module description are searched for the character `@` and the postfix of the `@` is used to lookup a parameter in the parameter dictionary, thus automatic assigning the referenced value. This automated assignment only works with module level tags and their values and does not search for a nested use in more structured tag values. Expending experiment description yield the one in `Step_1_SineHistoryGlobal.json` that is partly shown here:

```
{
    "ID":"Tutorial_Step_1_Sine",
    "nrD":300,
    "rSeed":12345,
    "parameter":
    {
        "domain":[0,1]
    },
    "modules":[
    {
        "ID":"InstanceGenerator",
        "collection":"Distribution",
        "domain":"@domain",
        "module":"Uniform"
    },

...

    {
        "ID":"LearningHistory",
        "collection":"Plotting",
        "module":"PlotLearningHistory",
        "samplingDomain":"@domain",
        "samplingSize":101,
        "legend":"Training Data",
        "input":
        {
            "xLearn":"InstanceGenerator",
            "yLearn":"InstanceLabeling",
            "ILS":"IncrementalLearningSystem",
```

```
            "GroundTruth":"InstanceLabeling"
        }
    }

}
```

Defining the domain of the experiment in this way is quite comfortable, but in this case unfortunately does not yield the freedom one might expect as the domain is used in the definition of the incremental learning system as well in the non-simple `approximator` tag and the used learning system does not support the global parameters, yet. All in all, we can define global parameters and can easily access them using the `@` prefix at module level tags, but using them in tags that are not at module level and thus highly module specific, requires the explicit support of the module.

## 2.2 Step 2: An advanced learning experiment

In this section we will let the learning system face some challenges picked from the list in the introduction and familiarize ourselves with different ways of passing parameters to modules. Furthermore, we learn how to group modules in order to encapsulate complex tasks based on basic modules and we take a look at the implementation on certain modules in order to see how the behaviour we observe at module level is mapped to python code. This will guide us to the overall structure of the PyUOSLib and the assumptions driving the framework.

### 2.2.1 Noisy learning data

The setting we envision in the advanced learning experiment is to learn a non-stationary target function from noisy data, but we will get there step by step. First we will expand the basic sine experiment form step one by adding noise to the labelling. This is done by inserting a noise module into the data flow from the instance labelling to the incremental learning system. The module we want to insert is described below:

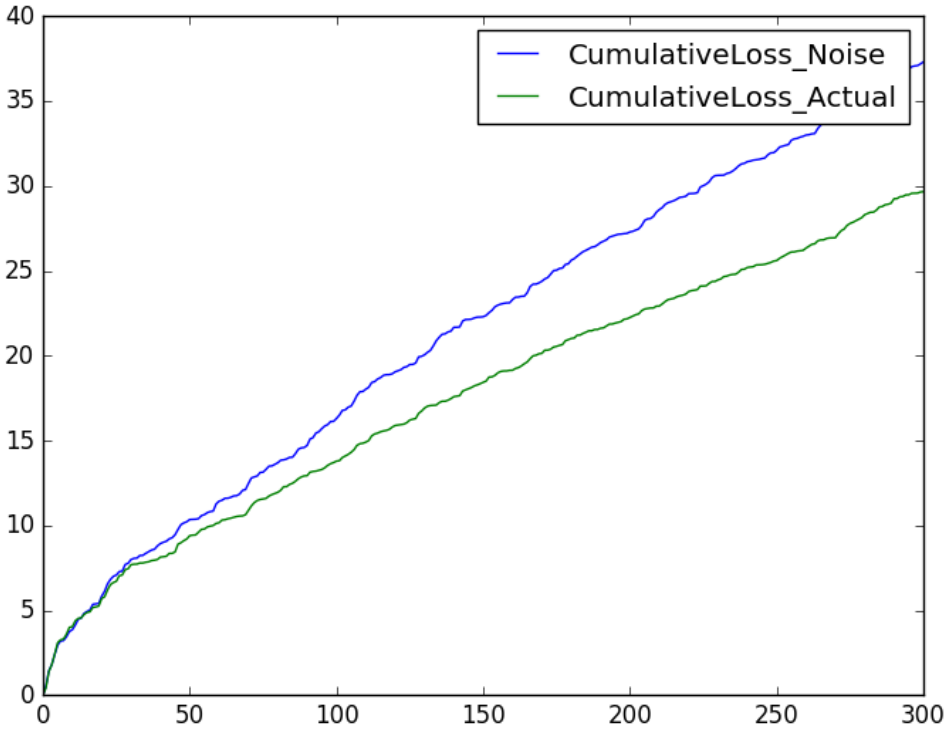
```
{
    "ID":"Noise",
    "collection":"AddNoise",
    "module":"Gaussian",
    "std":0.1,
    "input":
    {
        "value":"InstanceLabeling"
    }
}
```

The module adds a random number drawn according to a Gaussian distribution with zero mean and standard deviation 0.1 to the signal received in value and outputs the

sum. In this case it is obvious to feed the incremental learning system with the noisy data instead of the actual labelling because we want to learn from noisy data. But it is far less obvious whether we should measure the performance of the incremental learning system based on its ability to predict the noisy data or the actual labels. In many real world applications like filtering noisy sensor readings there is no ground truth and thus we do not have access to the actual labelling, thus the performance measurement based on noisy data gives a more realistic impression. But in this artificial setting as we do have access to the actual labels it is equally possible to measure the performance based on perfect data. So, ideally we do both and compare the different findings in order to see how the two measurements differ. This way we get more insights about the learning system and the performance measure as well. Thus, we duplicate the performance measure module and feed one with noisy data and one with the actual labels and plot both, which leads us to the next plotting module as the one we are using now is not suited for comparing different signals over time. Therefore we replace the `Plot` module by the following one:

```
{
    "ID":"Plot",
    "collection":"Plotting",
    "module":"PlotVsTime",
    "input":
    {
        "CumulativeLoss_Actual":"PerformanceMeasureActual",
        "CumulativeLoss_Noise":"PerformanceMeasureNoise"
    }
}
```

In contrast to the simple `Plot` module, the `PlotVsTime` module accepts any number of inputs and visualizes the development of the input values over all learning steps. It uses the key of each item from the input dictionary to name the different plots in a legend. At this point we have changed the basic sine learning scenario in different ways concerning the noisy labels, the performance measurement and the plotting. The resulting .json file is put in `Step_2_Noise.json` and executing it via `MPyUOSLib.py` should yield the following performance plot:




This plot shows that there is a difference between the two performance measures as the asymptotic slope of the cumulative loss based on noisy data is greater than the one based on the actual data, but essentially both measures capture the same overall course of the cumulative loss.

### 2.2.2 Non-stationary learning data

Now we have an experiment description to learn from noisy data, next we want to make the target function, i.e. the labelling, time variant in order to get the envisioned non-stationary learning setting. Making the target function non-stationary yields a complex module behaviour which we want to realize by combining different basic modules. In addition to that, we group the basic modules in order to encapsulate the complex behaviour provided by the basic modules. The `group` module in the PyUOSLib provides the frame for encapsulating different basic modules that together form a complex behaviour. A group behaves like a single module in the PyUOSLib but has its own internal data flow and module structure. Thus, each group definition contains a module list like the one we know from the experiment description. As each group behaves like a module in the PyUOSLib, it must provide an output for the module subsequent in the data flow of the experiment. The output definition of the `group` module is implicit as it uses the output of the last module of its module list as its own output. Thus, when designing the module list for a group one has to keep in mind the special role of the last module in the list as it determines the output of the whole group.
The non-stationary aspect we want to put into the target function is a frequency

drift. In fact we will expend the noise sine experiment by doubling the frequency of the sine from $2\pi$ to $4\pi$ over the span of 300 learning steps and continue the experiment with the doubled frequency for another 300 learning steps yielding a total of 900 learning steps for this non-stationary setting. In order to realize this kind of non-stationary behaviour of our target function we need a signal representing the frequency we require in each learning step, i.e. a signal that is constant at a value of $2\pi$ for 300 steps, then linearly increases to $4\pi$ over 300 steps and again is constant at $4\pi$ for 300 steps. Such a signal can be described using the `SignalBuilder` module as defined in the following module description:

```
{
    "ID":"SineFreq",
    "path":"Benchmark",
    "module":"SignalBuilder",
    "segments":[
        {"duration":300, "kind":"constant", "value": 6.28},
        {"duration":300, "kind":"linear",   "start": 6.28,
                                            "final":12.56},
        {"duration":300, "kind":"constant", "value":12.56}
    ]
}
```

The `SignalBuilder` module allows to define functions as a sequence of different simple segments. Each segment has a duration and a shape determined by its kind. There are three different shapes available: `constant`, `linear` and `halfcosine` (a cosine half-wave). The height of the constant shape is specified using the `value` tag, the non-constant shapes blend between the given `start` and `final` values. As each segment is defined independently from the rest, it is possible to build non-steady functions.

### 2.2.3 Passing parameters to modules

At this point we have a signal representing the course of the frequency we want to use and we need to plug it in the sine module for modulating its frequency. Up to this point we only used the required `x` input of the sine module, but the module has two additional optional inputs one for the phase and one for the frequency. The default value for these inputs are zero and $2\pi$ respectively. Using the signal builder output to feed the frequency input from the sine module, the sine module description becomes:

```
{
    "ID":"TargetFunction",
    "path":"Benchmark",
    "collection":"TargetFunction",
    "module":"Sine",
    "input":
    {
```

```
        "x":"argIn",
        "freq":"SineFreq"
    }
}
```

In order to understand the mechanism allowing us to have optional module inputs we need to take a look at the implementation of the Sine module:

```
class Sine(BasicProcessingModule):
    """
    Norm-based sine function.
    """
    def __init__(self, foot):
        default = {"freq":2*np.pi, "phase":0.0}
        BasicProcessingModule.__init__(self, foot, default)

    def __call__(self, x, freq=None, phase=None, index=0):
        if freq is None:
            freq = self.freq
        if phase is None:
            phase = self.phase
        return np.sin(freq*np.linalg.norm(x)-phase)
```

Important for us here is the argument list of the `__call__` member function. As we can see, the required input `x` is defined as a positional argument while the optional inputs `freq` and `phase` are keyword arguments with given defaults values. Here all default values are `None` and the following if statements check for this default value in order to replace it with an internal default. This internal default is given in the module initialization as we can see in the constructor, but the default values there are only active if the values are not defined in to module description in the .json file of the experiment. So, in the PyUOSLib there are three ways of passing parameters to a module:

- by *coding default values* in the module implementation,
- by specifing module parameters in the *module description* and
- by feeding *optional module inputs*.

These three way provide different kinds of flexibility as the implementation of the module is static, so are the parameters defined there. Specifying the parameters in the module description allows the experimenter to use different parameters for different experiments and thus yield flexibility along different experiments, but the parameters remain static for each experiment. Passing parameters as inputs to the module yields full flexibility, as every parameter may change in every single step of the experiment. The hierarchy of parameter passing methods follows their flexibility, so more flexible methods dominate less flexible ones.
Despite the parameter passing methods, the simple code snippet from above tells us many things about the way the PyUOSLib executes experiments. In order to

form its output, each module is called with a certain set of input arguments build from their input description. Looking at the rest of the code we see that the `Sine` module inherits from the `BasicProcessingModule`. In fact, every module in the PyUOSLib must inherit this class, as it provides the necessary interface that allows the `MPyUOSLib.py` to take care of the correct data flow specified in the experiment description. The initialisation of the module takes the input arguments `foot`, short for footprint, and `default`, which is an optional argument setting default value for module parameters. The footprint handed to the constructor here is the description of this very module given in the .json file defining the experiment. Thus, everything defined in the .json file to specify a module is available for initializing this module. Moreover, the `BasicProcessingModule` adds every item of the module description dictionary to the python object representing the module as a member variable filling up the missing parameters with the items of the default dictionary. Thus, there is no need to explicitly access the `foot` dictionary in order to set simple parameters defined in the module description.
The last thing we want to mention about the internal of the PyUOSLib here is the last input argument of the `__call__` function, namely the `index`. The `index` represents the time index of current learning step. This index allows e.g. the signal builder to select the current segment of the specified signal but can usually be ignored as in general the modules are designed to perform their task in a data flow irrespective of the elapsed time.

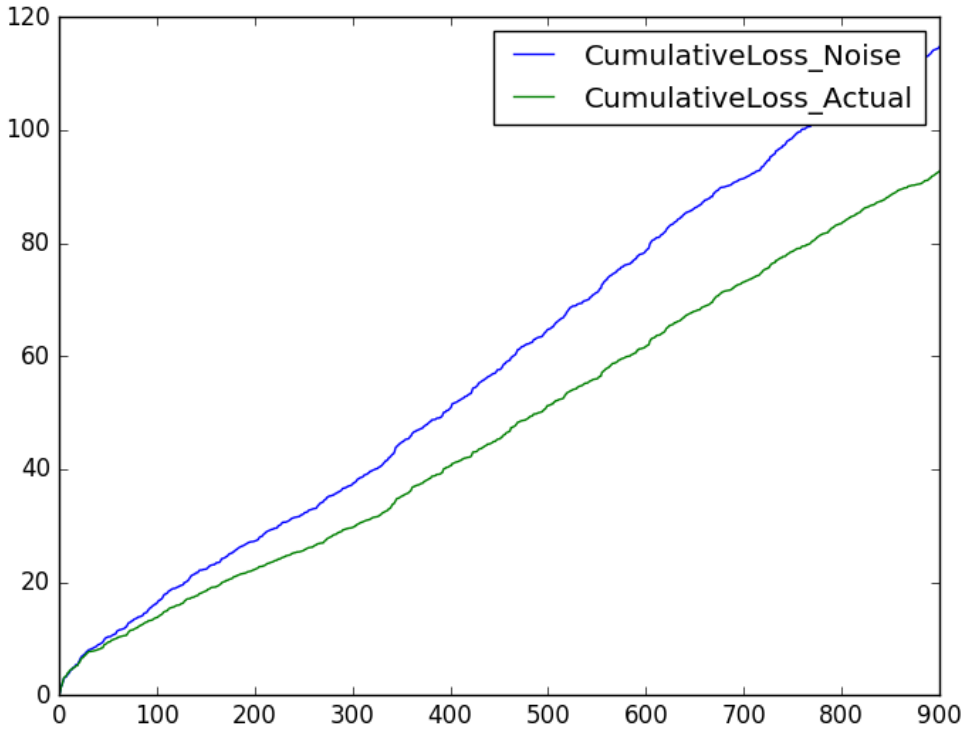
### 2.2.4 Grouping modules

The `SignalBuilder` and the `Sine` together form the non-stationary behaviour of the target function we envision and the composed complex behaviour provided by them is encapsulated by a PyUOSLib `group` module replacing the instance labelling of the simple learning expriment. The module description of the group we integrate in our experiment is shown below:

```
{
  "ID":"InstanceLabeling",
  "module":"Group",
  "input":
  {
    "argIn":"InstanceGenerator"
  },
  "modules":[
    {
      "ID":"SineFreq",
      "path":"Benchmark",
      "module":"SignalBuilder",
      "segments":[
        {"duration":300, "kind":"constant", "value": 6.28},
        {"duration":300, "kind":"linear",   "start": 6.28,
                                            "final":12.56},
        {"duration":300, "kind":"constant", "value":12.56}
```

```
      ]
    },
    {
      "ID":"TargetFunction",
      "path":"Benchmark",
      "collection":"TargetFunction",
      "module":"Sine",
      "input":
      {
         "x":"argIn",
         "freq":"SineFreq"
      }
    }
  ]
}
```

Usually it is not necessary to group different modules in order to encapsulate their behaviour, but when using a `PlotLearningHistory` plot it is mandatory to represent the actual labeling in one module allowing the `PlotLearningHistory` to uniformly sample different target function, even non-stationary ones. The complete description for the advanced learning experiment of learning a non-stationary target function from nosiy data is put in `Step_2_Non-stationary.json` and should yield the following performance plot:



## 2.3 Step 3: Analysing different parameter settings

The experiments in step one and two show us that the learning system we are using is prone to noise as the asymptotic slope of the cumulative loss increases drastically when switching to noisy labels. But from our two observations one without and one with noise we cannot tell if the noise has a linear or non-linear impact on the performance. Therefore, we what to perform a set of experiments by varying the noise level and comparing the cumulative loss. As we already know, adding noise to the label does not change the progression of the cumulative loss in a fundamental way, thus in order to reduce the whole development of the loss to one value we can just look at the final cumulative loss as it allows us to compare different settings in an abstract way allowing us to focus on the impact of the noise level. So we next, want to perform the simple experiment from step two with different noise levels, i.e. we what to vary the standard deviation of the Gaussian distribution used to induce noisy labels. In terms of a data flow we are looking for a module we can feed the standard deviation we want to test and outputs the final cumulative loss. Thus, the module we envision is an experiment on its own. The PyUOSLib supports this perspective by allowing to feed experiments with inputs and to nest them.

### 2.3.1 Nested experiment descriptions

It requires only small changes to turn the simple learning experiment with noise into the module we need for our nested experiment. First of all, we need to mark the experiment as a `ModularExperiment` module, making it accessible by others as a PyUOSLib module. Secondly, the experiment seen as a module needs some input. In this case, it is the standard deviation. Next, we need to get rid of the plotting as we no longer want to observe the whole development of the learning process but focus on the final cumulative loss. Finally we need to define the output of the experiment. Here again the output of the experiment is determined implicitly as the output of the last module of the module list, following the behaviour we already know from the `group` module. The final module description of the parametrized experiment looks like this:

```
{
    "ID":"ParameterizedExperiment",
    "module":"ModularExperiment",
    "nrD":300,
    "rSeed":12345,
    "input":
    {
        "GaussianStd":"StdIterator"
    },
    "modules":[
        {
            "ID":"InstanceGenerator",
            "collection":"Distribution",
            "module":"Uniform"
```

```
},
{
    "ID":"InstanceLabeling",
    "path":"Benchmark",
    "collection":"TargetFunction",
    "module":"Sine",
    "input":
    {
        "x":"InstanceGenerator"
    }
},
{
    "ID":"Noise",
    "collection":"AddNoise",
    "module":"Gaussian",
    "input":
    {
        "value":"InstanceLabeling",
        "std":"GaussianStd"
    }
},
{
    "ID":"IncrementalLearningSystem",
    "path":"IncrementalLearning",
    "collection":"IncrementalLearningSystem",
    "module":"CompoundLearningSystem",
    "approximator":
    {
        "kind":"TensorExpansion",
        "inputs":[{"kind":"GLTlinear",
                   "linspace":[0,1,11]}]
    },
    "learner":{"name":"PA"},
    "input":
    {
        "xLearn":"InstanceGenerator",
        "yLearn":"Noise"
    }
},
{
    "ID":"PerformanceMeasure",
    "collection":"PerformanceEvaluation",
    "module":"CumulativeLoss",
    "input":
    {
        "target":"Noise",
```

```
                "prediction":"IncrementalLearningSystem"
            }
        }
    ]
}
```

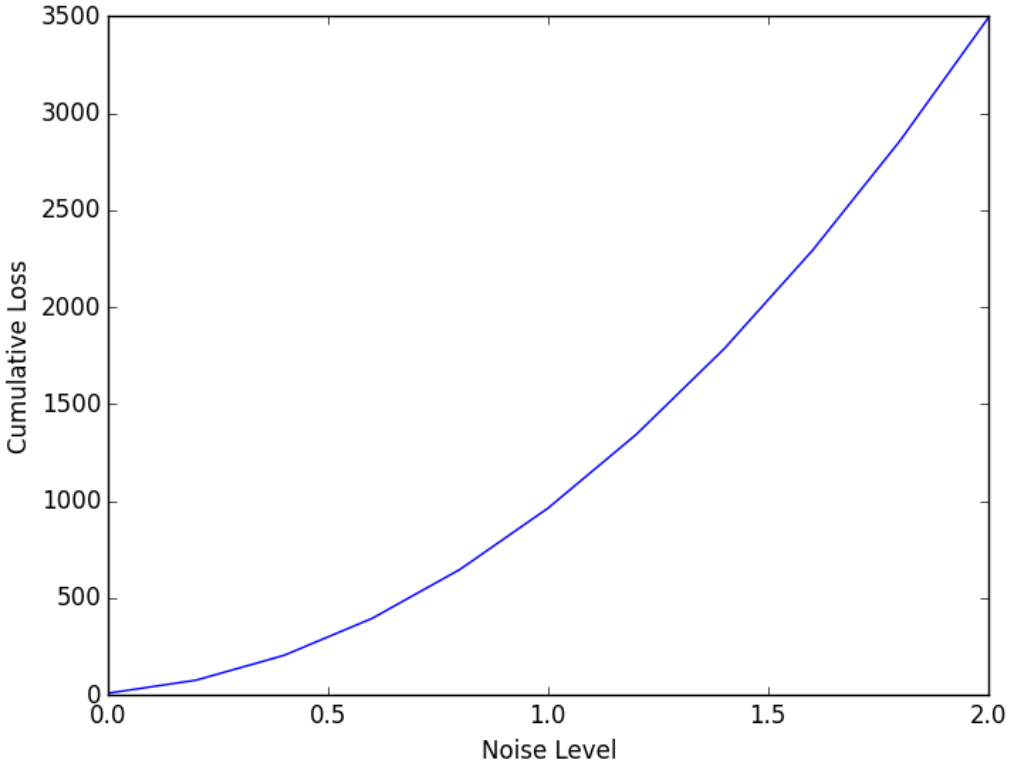
Having this module as the main part of the experiment we are heading for, we only need to feed this module with the noise levels we want to test using a `SignalBuilder` and gather the results in a `Plot`. Here, we want to test 11 noise levels ranging from 0.0 to 2.0 yielding a `SignalBuilder` module description like the following one:

```
{
  "ID":"StdIterator",
  "path":"Benchmark",
  "module":"SignalBuilder",
  "segments":[
    {"duration":11, "kind":"linear", "start":0.0,"final":2.0}
  ]
}
```

For plotting we again use the `Plot` module as it allows us to separately define the values to be plotted on the x- and y-axis and label them accordingly. The complete `Plot` module description is given in the following:

```
{
    "ID":"Plot",
    "collection":"Plotting",
    "module":"Plot",
    "xlabel":"Noise Level",
    "ylabel":"Cumulative Loss",
    "input":
    {
        "x":"StdIterator",
        "y":"ParameterizedExperiment"
    }
}
```

Putting all these three modules together yields the complete experiment description for the nest experiment setting. It allows us to test the impact of different noise levels to the performance and is filed in `Step_3_NestedExperiment.json`. The resulting plot should look like the following one:




The development of the cumulative loss regarding the noise level looks like a parabola and thus indicates a quadratic relationship. But as the loss itself is based on the squared difference between given `target` and `prediction`, the impact of the noise level here is only linear and this linear impact appears as a parabola due to the quadratic nature of the performance measure used here.

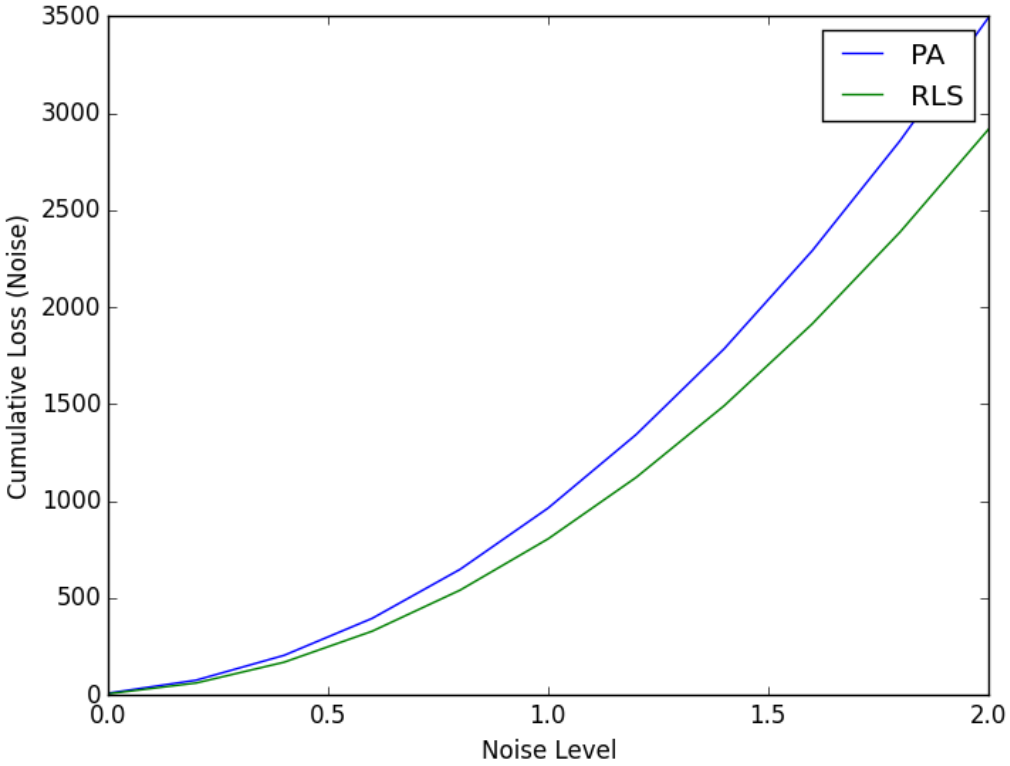
### 2.3.2 Comparing different learning algorithms

When comparing different learning algorithms it is in general necessary to define a parametrised experiment for each approach you want to test, as the variety of learning algorithms is hard to unify in one learning module. Here, we will compare two different on-line learning algorithms in order to keep thing simple. The one algorithm we use is the PA (Passive-Aggressive) approach we already used in this tutorial and we will compare it to RLS (Recursive Least Squears). We will test if RLS is less prone to noise than PA and thus can mainly reuse the experiment description in `Step_3_NestedExperiment.json`. The first thing we do is to double the `ParameterizedExperiment` description in order to have two modules receiving the same noise level and performing the same experiment. Obviously, the two modules need different `ID` so we name the original one `ParameterizedExperimentPA` and its copy `ParameterizedExperimentRLS`. In the copy we also change the `learner` from PA to RLS and thus change the learning algorithm used in this experiment.
Now, we have two different `ModularExperiments` providing us with data about different noise levels and we want to plot the data. And here we face some kind of problem as the `Plot` only accepts one input for the y-axis and the `PlotVsTime`

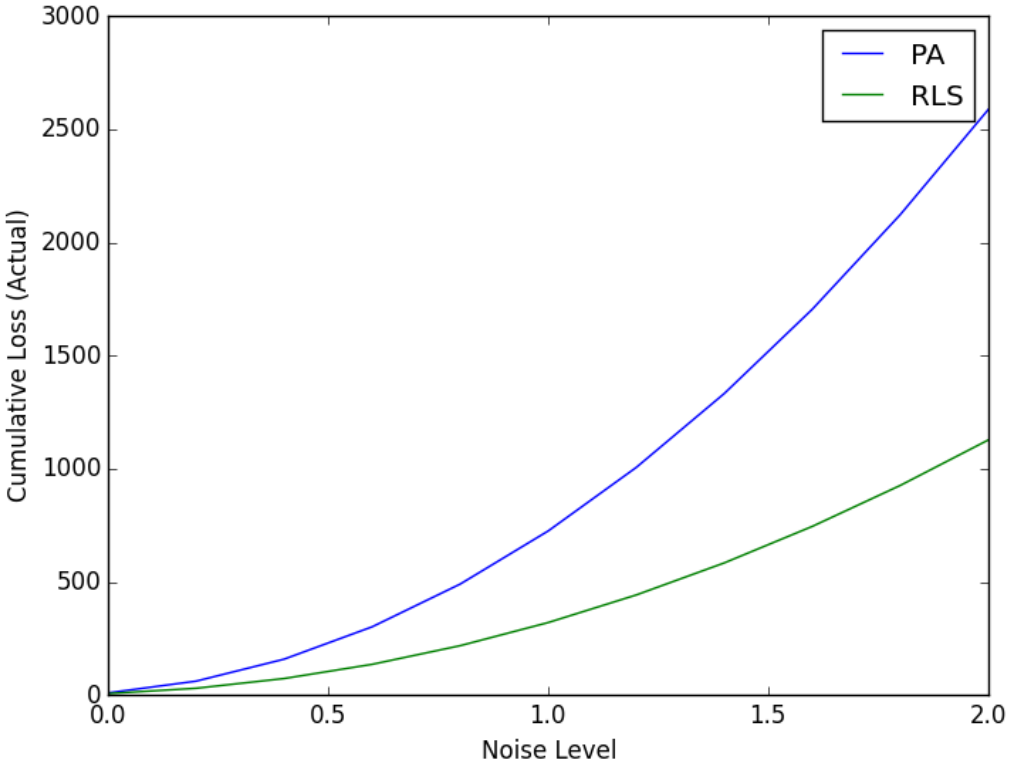
module does not take value for the x-axis. We can use the `Plot` module as it allows to plot multiple line for vector valued input at `y`, but we need to group the single results from the experiment modules in one vector before feeding this vector to the plotting. The `BusCreator` module allows us to do this and doing so leads us to a tricky and nasty detail about to cooperation between Python and PyUOSLib. To see the point we first need to take a look at the implementation of the `BusCreator` module from the basic collection:

```
class BusCreator(BasicProcessingModule):
    """
    Group many scalar inputs to one vector
    """
    def __init__(self, foot):
        BasicProcessingModule.__init__(self, foot)
        self.nrIn = len(self.input)
        self.output = np.zeros(self.nrIn)


    def prepare(self, antecessor):
        self.order = self.input.keys()
        self.order.sort()

    def __call__(self, index=0, **argIn):
        out = np.zeros(len(self.input))
        for i in range(self.nrIn):
            out[i] = argIn[self.order[i]]
        return out
```

The point here is the different treatment of ordering for lists and dictionaries in Python. When defining a list in a .json file we can be sure to have the exact same ordering in the python list object representing our list. On the contrary, a dictionary is a set of items and has no natural ordering and iterating the items of a dictionary may yield any order but the one we had in the experiment description. Using dictionaries to represent the inputs in the PyUOSLib is straight forward as it fits to the way Python handles its input arguments and, as indicated by the `**argIn` field, the `BusCreator` allows the usage of any number of inputs. But as these inputs are represented as a dictionary we cannot simply iterator these arguments. In fact, we need to somehow make the intended ordering of the inputs explicit. The `BusCreator` module uses the ordering of the key entries to fix this order. Most of the time this delicate difference in python regarding the handling of lists and dictionaries is ignorable, but it is good to know.
Heading back to our topic of comparing different learning algorithms, we now directly get to the experiment description in `Step_3_CompareLearningAlgorithms.json` and the resulting plot:




As it turns out, RLS is in fact less prone to noise than PA, but here we compare the prediction performance for the noisy samples. We cannot expect any method to predict the noise as we want to learn the underlying function, i.e. the actual labelling, and not the noise. Thus, we should rerun our experiment and measure the performance based on the prediction quality for the actual labelling. The corresponding experiment description is filed in `Step_3_CompareLearningAlgorithms_Actual.json` and yield the following plot:




This simple comparison of two different ways of measuring the performance of a learning system is meant to sensitise to the following guiding questions in designing experiments:

- What do I want to test?
- How do I want to test that?
- What are my expectations?

# 3 Developing modules

At this point we are able to define and run experiments using the PyUOSLib and we have seen some of its internal mechanics as well as some basic modules that are already part of the PyUOSLib. Now we want to build our own module, say an additional learning algorithm namely a windowed NN (nearest neighbour) variant using only the most recent learning data. But first thing first, we are just starting a new project and thus want to group our files in one folder. Therefor, we create a new subfolder in `Modules` named `MyProject`. We place an empty `__init__.py` file in this newly created folder in order to make it visible for python. Then we create a file named `MyCollection.py` inside our folder.

## 3.1 A nearest neighbour learning algorithm

Let's fill our first collection with the NN learning algorithm. As experts in the PyUOSLib we know that we don't need to start from scratch and we need to inherit `BasicProcessingModule`. So we know that there is a interface for incremental learning systems as we already use these system in our first steps. We also know that it is a good idea to inherit this interface as it supports the abstraction we want to use. But, how to import this class and what functions does this interface actually provide? The import is python related and as we run all of our experiments using the `MPyUOSLib.py` all imports go relative to the folder containing it, thus in order to import the `IncrementalLearningSytem` we use:

```
from Modules. \
IncrementalLearning. \
IncrementalLearningSystem import IncrementalLearningSystem
import numpy as np
```

The numpy import here is not mandatory but common scenes throughout the framework. The interface the incremental learning system requires is determined by two functions:

- `evaluate(self, x)`

  Predict the label of the instance $x$.

- `learn(x, y)`

  Incorporate the learning date $(x, y)$

Now the only thing we need to do is to specify how we evaluate our approximation and how we learn and our incremental learning system is ready to use. This yields the following raw NN class:

```
class NearestNeighbour(IncrementalLearningSystem):

    def __init__(self, foot):
        IncrementalLearningSystem.__init__(self, foot)

    def prepare(self, antecessor):
        pass

    def evaluate(self, x):
        return 0.0

    def learn(self, x, y):
        pass
```

Next, we define how to initialise, prepare, evaluate and learn our NN learner. As a lazy learning method NN allows us a lazy initialization, too:

```
def __init__(self, foot):
    default = {"windowsize":50}
```

```
    IncrementalLearningSystem.__init__(self, foot, default)
    self.index = 0
```

We only set the default size of the window we want to use and initialize our internal index. All other initialisations are done using the `prepare` function in order to be able to react to the actual inputs the module will face during the evaluation:

```
def prepare(self, antecessor):
    self.nrIn = np.size(antecessor["xLearn"].output)
    self.windowX = np.zeros([self.windowsize, self.nrIn])
    self.windowY = np.zeros([self.windowsize])
```

This way, we can set up the correct size for the numpy array we use to represent our window. Before we define the evaluation we take care of the learning because it is by far more easy:

```
def learn(self, x, y):
    idx = np.mod(self.index, self.windowsize)
    self.windowX[idx,:] = x
    self.windowY[idx] = y
    self.index += 1
```

The learning just implements a circular buffer using an internal index to keep track of the learning steps and indexing the corresponding window elements. The evaluation is nearly as easy:

```
def evaluate(self, x):
    maxIdx = np.min([self.windowsize, self.index])-1
    if maxIdx<0:
        return 0.0
    else:
        dist = ((self.windowX[:maxIdx+1]-x)**2).sum(1)
        return self.windowY[np.argmin(dist)]
```

We use the index to track the size of the window in order to avoid accessing elements not yet learned and output the nearest neighbour based on the euclidean distance. Last but not least, we define the `reset` function and take care to rebuild a valid initial state as we first did using the initialisation and preparation:

```
def reset(self):
    self.windowX = np.zeros([self.windowsize, self.nrIn])
    self.windowY = np.zeros([self.windowsize])
    self.index = 0
```

Now our nearest neighbour learning algorithms is ready to use and we should compare its behaviour to the other learning methods we already saw in this tutorial.

## 3.2 A flexible arithmetic module

The learning module we developed in the previous section has a static interface fully described by the `IncrementalLearningSystem`. In this section we will build

an arithmetic module with a variable interface, thus an interface defined in the experiment description not in the module implementation.

```
class AddSubArithmetic(BasicProcessingModule):
    def __init__(self, foot):
        BasicProcessingModule.__init__(self, foot)

    def __call__(self, index=0, **argIn):
        out = 0.0
        for k,v in argIn.iteritems():
            if k is '+':
                out += v
            else:
                out -= v
        return out
```

Important for us here is the `**argIn` dictionary gathering all keyword arguments given to call the function, but excluding the mandatory index argument as it is not part of the module description in the .json file experiment description. Thus, the `**argIn` dictionary contains all the inputs we defined in the module description named by the key we use there. Here, we use the keys to determine the sign of the corresponding input to the overall sum. This allows us to select in the module description whether this module calculates the sum or difference between any number of inputs.
When combining different arithmetic operations in one module like this, make sure that they are commutative as the ordering of the input description is not preserved when iterating the dictionary items.