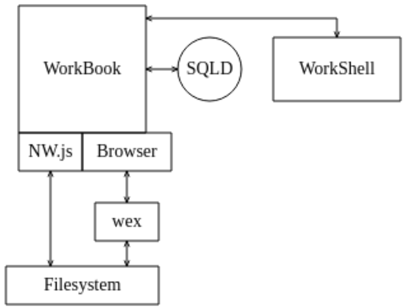
Maschinelles Lernen und Datenanalyse

In der Mess- und Prüftechnik PD Stefan Bosse

Universität Bremen - FB Mathematik und Informatik

WorkBook, WorkShell, and Dataflow Graphs

Introduction to Dataflow Graph Architectures using the WorkBook



Machine learning

Preparation

0. Experiments providing measuring and meta data
1. Input feature selection
2. Data transformation
    - Functional, reduction
    - Format, data type
3. Statistical Analysis
4. Data splitting
    - Training sample partition
    - Test sample partition

Machine learning

Model Training

5. Selection of model, defining model parameters
   - Static parameters: model structure
   - Static parameters: operational parameter (learnign rate, etc.)
6. Iterative training of model
   - Adaptation of dynamic parameters (functional parameter or structure)
7. Calculation of the prediction/inference errors:
   - Training data
   - Test data
8. Reconfiguration of static model and algorithm parameters, go to 6

Machine learning

Functional and Dataflow Graphs

> The entire data processing architecture can be mapped on a dataflow graph (DFG)

- A DFG consists of functional (stateless) and procedural (state-based) nodes with:
  - Input ports (input variables) ***i***
  - Output ports (output variables) ***o***
  - Operational ports (state- and event-based methods of a node) ***op***

$$f_n\left(\vec{i}\right) : \begin{cases} \vec{i} \rightarrow \vec{o} & op_1 \\ \vec{i} \rightarrow \vec{o} & op_2 \\ .. & .. \\ \vec{i} \rightarrow \vec{o} & op_k \end{cases}$$

$$p_n\left(\vec{i}, \sigma\right) : \begin{cases} \vec{i} \times \sigma \rightarrow \vec{o} \times \sigma & op_1 \\ \vec{i} \times \sigma \rightarrow \vec{o} \times \sigma & op_2 \\ .. & .. \\ \vec{i} \times \sigma \rightarrow \vec{o} \times \sigma & op_k \end{cases}$$

$$DFG(\vec{x}) : \vec{x} \rightarrow f_1 \rightarrow f_2 \rightarrow \begin{cases} f_3, .. \\ f_4, .. \\ f_5, .. \end{cases} \rightarrow f_m \rightarrow \vec{y}$$

There are sequential and parallel data paths. An output port can be connected to multiple input ports → Data split. There are join nodes (demultiplexer or aggregators).

There is a separation of computation and communication.

Such DFG architectures can be easily parallelized and distributed (Web) using IP-based communication channels (e.g., using WebSockets)!

DFG Node Blocks

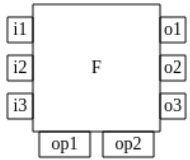


Fig. 1. Data interface of a functional DFG node with input, output, and operational ports

- DFG nodes can be connected:
    - Output ports are connected with input ports ⇒ **Directed data channels**
    - Output ports can be connected to operation ports ⇒ **Directed data event channels**

DFG Node Library

There is a object-oriented class library **L** that provides class constructor functions implementing DFG nodes.

- Nodes are instantiated from the class constructor with a set of individual parameters
  - Names (identifies) of input and output ports
  - Data formats and types
  - Functional parameters
  - Visual parameters

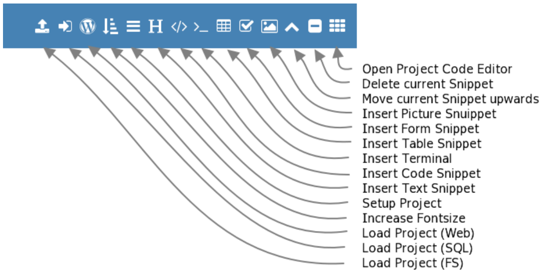
WorkBook

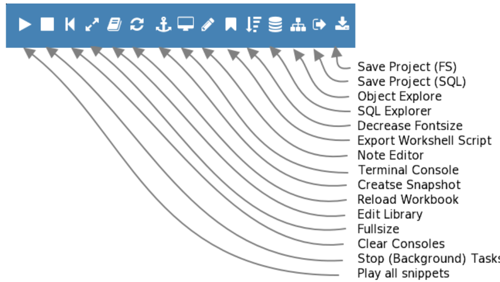
The WorkBook is a Web browser application consisting of HTML/CSS content and JavaScript code.

- The WorkBook can be executed in any Web browser or using node-webkit (nwjs)
- The WorkBook program flow is structured in snippets:
    - Code snippets
    - Text snippets
    - Table and form snippets
    - ...
- A WorkBook project consists of a sequence of snippets, code, and data.

WorkBook

There is a main button toolbar providing the core operations to compose, control, and exchange WorkBook projects (JSON data):





Code Snippets

A code snippet is initially executed in the main JS loop. A code snippet consists basically of a code editor and an output console.

- If a code snippet performs long computations, the GUI is maybe not reactive!
- A code snippet should use only local variables.
- Local variables can be exported to a shared context (shared among code snippets)
- Shared context variables can be improted
- There are different fields and buttons in a code snippet providing specific operations
    - Hiding/Showing code snippets (full, editor, console)

Code Snippets

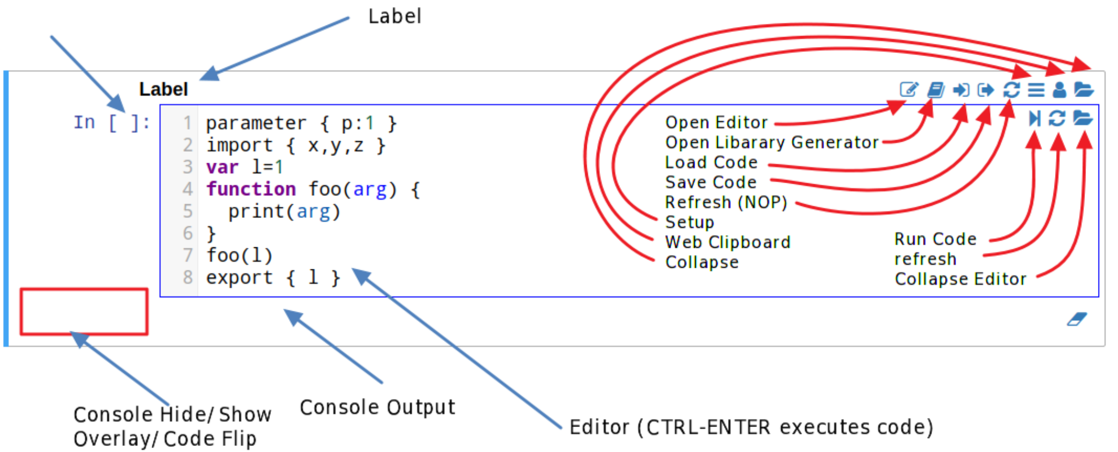


Fig. 2. Control of a code snippet

Code Snippets

A code snippet support to display modes:

1. Editor + Console
2. Overlay control + Console

> To enable overlay mode, open the set-up menu and enter a value ≠ "false" in the overlay field (e.g., 1 or true) **and** flip the display mode with the field in the lower left corner (see previous figure).

The overlay view contains three basic buttions (Run code, stop backrgound tasks, and clear console) on the left side, import and export fields, and an optional parameter table on the right side.

Code Snippets

Parameter Tables

A parameter table consists of key-value rows. The value can be changed by clicking in the cell. By right clicking a context menu can be opened.

- Parameters are defined in the code by using a non-standard parameter statement. Any value type can be added, and choice lists (via right click context menu) are supported by a second underscore parameter providing the choices.
- Parameters can be accessed as free variables (e.g., `p1` in the following example).
- Parameters can be changed by accessing the *parameter* object.
- Parameter values are saved!
- An event handler for parameter changes can be installed (see example)

Code Snippets

```
parameter { p1:'v1', _p1:['v1','v2','v3'], p2:0, p3:{a:1,b:2} }
...
parameter.p2=100;
parameter._on = (key,val) => { print(key+' changed:'+val }
if (p1=='v1') { .. }
```

- Each time a parameter was changed (overlay mode activated and visible), the event handler will be called. The changed parameter can be forwarded to other functions.
- The right-click context of each editable cell menu provides value choice lists, a generic text editor for comfortable editing of cell content, and a filesystem explorer to include file paths.

Code Snippets

Import and Export

Parameter setting, import, and export statements in code snippets are **not valid JS syntax** (proprietary)!

```
import { a,b,c,d ..}
var e,f,g,h,..

export { e,f,g,h,.. }
```

Class Library

To provide easy anc convenient access to nuemrical and ML modules, there is a class library providing:

- There are pre-configured node classes for data sources, data transformation, data display, and ML (and many more)
- Node constructors of a class library can be generate dby opening the library form (in the editor button bar)
- After node set-up, code is generated in the current selected code snippet creating an instantiated object and updating or creating import and parameter statements
- Commonly, the node objects are nodes of a Dataflow Graph

Dataflow Graphs (DFG)

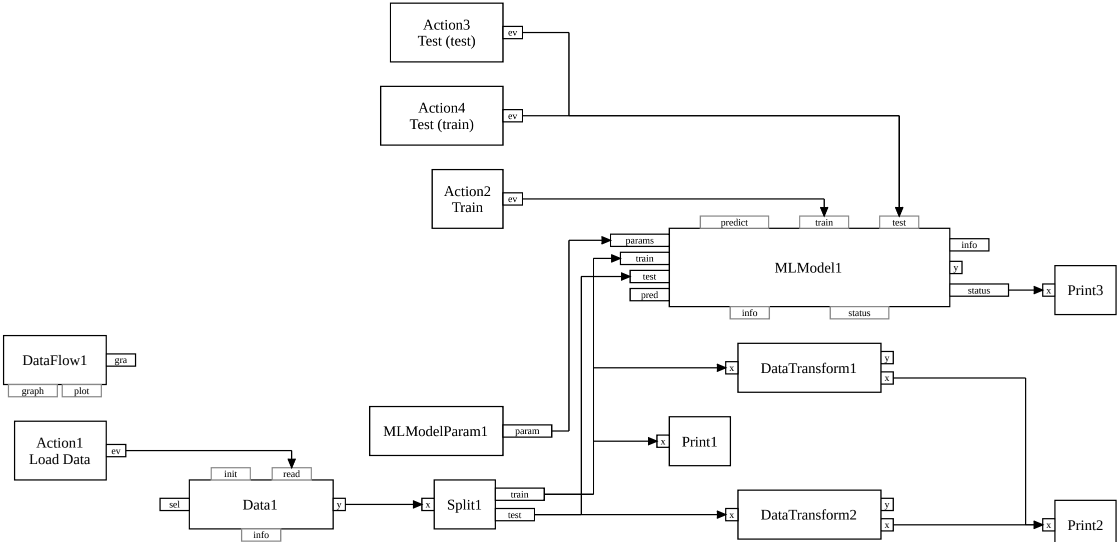


Fig. 3. Example of a DFG composed of nodes generated by the class library

DFG Nodes

Actions

Actions provides (parameterizable) button events. The button event (via node output or action callback handler) can be connected to other nodes triggering the execution of a node method (normally preceded with a "~" character to avoid ambiguities with input port names)

```
import { Action, dataNode }
var actionLoadData = new Action({
  "label": "Load Data",
  "action": () => {
    status('Loading data table ...')
  },
  arguments:['*'],
})
actionLoadData.output(dataNode,'~read')
// === dataNode.read.apply(dataNode,arguments) == .read('*')
```

DFG Nodes

Data Source

Currently only SQL access via sqld RPC is provided.

```
import {Data}
var data = new Data("sql",{
  "url": "localhost:9999",
  "database": "Iris",
  "table": "iris1"
})
await data.init()
status(inspect(await data.info()))
// data.read("*",index)
// data.input("*",index)
// data.output(node)
```

DFG Nodes

Data Channels

- DFG nodes can be connected by (hidden) channels (uni-directional)
- A channel between an output port of node *A* connected to an input port of node *B* is created by calling the *output* method from node *A*:

```
nodeA.output(nodeB,xindex?,yindex?)
```

- The optional *xindex* argument selects the input port of the destination node *B*
- The optional *yindex* argument selects the output port of the source node *A*
- If *A* and *B* have only one output and input port the index selectors can be omitted.

DFG Nodes

Data Formats and Types

Data can be represented in different formats (data types). Assuming data tables (columns represent the input feature and output target variables, rows represent different samples), there are two major formats:

`[[]]`
: Array tables (arrays of arrays). Each row of the outer table array is an array containing values of the feature and target variables. Each column entry is accessed by a numeric index (starting with 0).

`[{}]`
: Record tables (arrays of records). Each row of the outer table array is a record containing values of the feature and target variables. Each column entry is accessed by their attribute name.

DFG Nodes

Data Splitter

A data splitter is used to create randomly selected partitions from the full data set, e.g., a data table. In ML, there is commonly a training set and a test set. The training set is only used for the model training process, the test set for evaluation of the model.

```
import {Split,dataNode}
parameter {ratio:[0.5,0.5]}
var splitData = new Split({
  "input": "[{}]",
  "outputs": [
    "train",
    "test"
  ],
  // "ratio": [0.5,0.5],
  "random": true,
  parameter:parameter,
})
dataNode.output(splitData)
```

DFG Nodes

Data Transformation

Data transformation is used to convert between data formats and to apply normalization/scaling (optionally).

```
import {splitData, DataTransform, Print}
var datatransformTrain = new DataTransform({
  "input": "[{}]",
  "output": "{x:[[]],y:[]}",
  "attributes":["length","width","petal_length","petal_width"],
  "targets":["species"],
  "inputs": [
    "x"
  ],
  "outputs": [
    "x",
    "y"
  ],
  "filter": (a) => { return a }
})
```

DFG Nodes

ML Model Parameters

An ML model and its algorithms are parametrized. There are static and dynamic parameters. Static parameters define the structure of the model (e.g., layer of an ANN or the polynom degree of a function, or optimization parameters like the learning rate), and dynamic parameters are those that are optimized by the trainer algorithm. Parameters are set using a editable table and forwarded to the ML model implementation as data.

```
import {MLModelParam,mlmodelDT}
parameter {
  features:["length","width"," petal_length","petal_width"],
  target:["species"],
  algorithm:"id3"}
var mlparamDT = new MLModelParam("c45",{
  "parameter": parameter,
  "display": false,
  "label": "C45/ID3"
})
mlparamDT.output(mlmodelDT,'params')
```