**Towards a flexible component-based architecture of machine learning software**

**Marcin Wojnarski**

**Warsaw University, Poland**

## The Principles

1. Loosely typed data representation:
   one base class for all kinds of data samples;
   type of data checked at run-time by down-casting.
2. One base class for all data processing algorithms, of any type.
3. Each algorithm implemented as a *component*.
   It can execute itself, without external control, because it knows where to find data to be processed.
4. Components are linked together into a *Data Processing Chain* – the „backbone" of the experiment to be executed. Execution is started by calling one method on the chain head. No other involvement of the caller is needed during execution, the chain executes itself.
5. Data passed between components sample-by-sample.

## Architecture of WEKA in brief

- Classifier:
  - buildClassifier(Instance**s**) : void
  - classifyInstance(Instance) : double
  - distributionForInstance(Instance) : double[]
- Clusterer:
  - buildClusterer(Instance**s**) : void
  - clusterInstance(Instance) : int
  - distributionForInstance(Instance) : double[]
- Filter:
  - input(Instance), batchFinished(), getOutputFormat(), output()
  - useFilter(Instance**s**, Filter) : Instance**s**
- Loader:
  - getDataSet() : Instance**s**
  - getNextInstance() : Instance
- Estimator, Associator, Generator, …

## Loosely typed data representation

```
struct Sample
{
    SampleData* data;
    Label* label;
    Label* decision;
}
```

- SampleData descendants:
  - NumericData
  - NominalData
  - ImageData
  - …
- Label descendants:
  - ClassLabel
  - NumericLabel
  - PositionLabel
  - ListLabel
  - …

## The Component

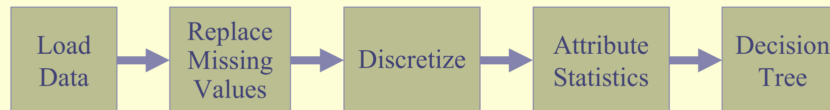
```
class DataProcessor
{
public:
    void Open();
    Sample* GetNext();
    void Close();
 protected:
    DataProcessor* source;
}
```

- Extremely simple interface: „get next portion of data”

## One base class for all algorithms

- *DataProcessor* interface can handle all types of data processing:
  - classification, regression, clustering, density estimation, …
  - choice of a subset of attributes / samples
  - transformation of attribute values (discretization, normalization, …)
  - gathering of statistics (untouched samples are passed further)
  - for images: rescaling, cutting subwindows, filtering, …
  - loading data from disk / database / …
  - generation of synthetic data
  - data materialization
  - random shuffling of samples, splitting into train/test sets, …

## Data Processing Chain




- Building plan of an experiment is separated from running it
- DPC is like a „program” in a very high-level language
- Executing DPC is very simple: „push the button”
- It's easy to make a change in DPC and run again

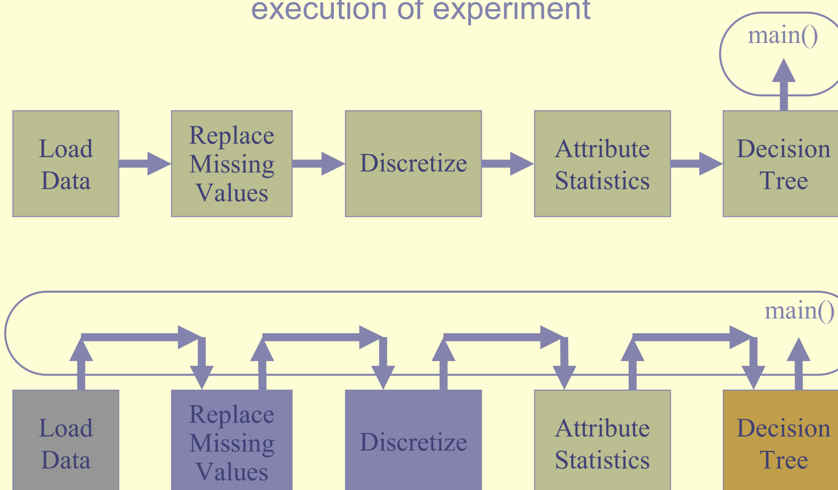
## Experiment

- Preparation of the experiment:
  - allocation of objects for each step of the experiment
  - setting parameters of the objects (e.g. no. of training epochs for NN)
- [Loading instead of preparation]
- [Saving the experiment]
- Execution of the experiment:
  - calculation of input data for the next object
  - invoking appropriate method on the object
- Saving the result of experiment (trained decision model, together with accompanying algorithms, e.g. preprocessing)
- Loading the result of experiment
  (e.g. to test on new data; to embed in a committee of systems, …)

# Experiment

- Preparation of the experiment:
  - allocation of objects for each step of the experiment
  - setting parameters of the objects (e.g. no. of training epochs for NN)
- [Loading instead of preparation]
- [Saving the experiment]
- Execution of the experiment:
  - calculation of input data for the next object
  - invoking appropriate method on the object
- Saving the result of experiment (trained decision model, together with accompanying algorithms, e.g. preprocessing)
- Loading the result of experiment
  (e.g. to test on new data; to embed in a committee of systems, …)

in the proposed architecture this is the only part which requires significant effort to implement



## Memory management

- Some experiments are memory-intensive: large data sets to be processed, e.g. images
- Trade-off: simple architecture vs. efficient memory usage
  - every processing step may create another copy of the data set
  - data may be efficiently stored in some problem-specific representation → architecture becomes messy
- Solution: passing samples one-by-one between processing steps → DataProcessor has an *iterator interface*
- Full control over memory usage: materialization can be put in any place of Data Processing Chain

## Case studies

- AdaBoost cascade for object detection in images:
  - don’t know in advance how many training images to generate
  - large amounts of data, memory issues
- Train & Test, cross-validation, classification of new data

## Summary

- Advantages of the proposed architecture:
  - can handle arbitrarily large volumes of data
  - easier to implement a new experiment
  - easier to run a series of similar experiments
  - easier to write new components (only one base class)
  - easier saving/loading of experiment: plan and result
- Disadvantages:
  - type correctness of the experiment is checked at runtime, not compile time
  - each component must be aware of what types of data it can handle; type of data must be checked explicitly by downcasting
  - in some cases passing data sample-by-sample may be slower than passing all samples at once
- My implementation of this architecture:
  currently aprox. 10 subclasses of DataProcessor

Thank You