STATG019 – Selected Topics in Statistics 2018

Lecture 4

Modelling & Workflow API design for Stats/ML Modelling Toolboxes

Dr Franz J. Király

Key requirements to an ML toolbox

Model interface: provide access to a wide class of models

e.g., OLS, support vector machines, neural networks, etc

Fitting and prediction: simple interface given train/test data

Settable hyper-parameters: easily accessible and changeable

This should be similar for all classes and kinds of models

Model tuning & composition: grid-tuning, ensembling, pipelines

Specification of tuning/composition parameter same for all models

Exposing meta-model parameters as hyper-parameters of result

Model validation & evaluation: estimation of generalization loss

for standard loss metrics and re-sampling based validation schemes

Running of benchmarking experiments including all the above

User/workflow interaction: experiment set-up and reporting

All enablers of reproducibility and scientific transparency!

Object oriented API design for statistical modelling strategies

Functions in **Programming** | **Mathematics**

outputY <- functionF(inputX)

$$f : \mathcal{X} \to \mathcal{Y} \qquad y = f(x)$$

Intuition:

Input x **inputX** → **Function** *f*, **functionF** → **Output** y **outputY**

every function has a specific **input/output signature**, for example:

inputX is of type **string**

$$\mathcal{X} \subseteq \mathbb{R}^2$$

functions can have **parameters** such as

functionF(input1,input2,paramN)

$$f_n : \mathcal{X} \to \mathcal{Y}, \ n \in \mathbb{N}$$

functionF(1,2,3) runs the function body code with **1** substituted for **input1** etc

When to use functions:

for repeated execution of code blocks in different places

and/or for varying settings of the input parameter values

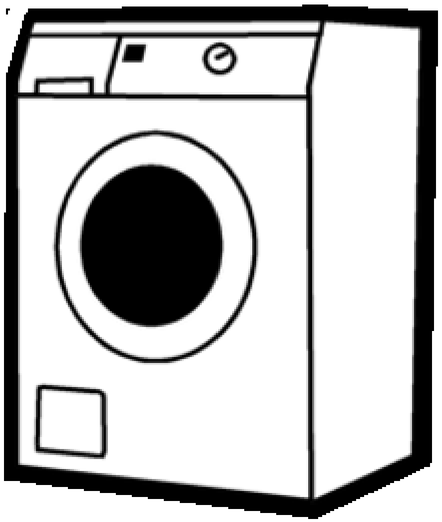
Object orientation: informal overview

bundles multiple code functionality in a single „object“

Unified interface

user-facing functionality and properties in one place

WashMachine.fill(clothes)   **WashMachine.run(1h)**



Encapsulation

function-specific code internal, hidden from user

WashMachine.run = function(time, mode = „wash“){

flow <- 5; spin <- 10; wait(1min); time <- time – 1min etc

drain <- 5; wait(1min); WM.reset() }

When to use object orientation:

to model a thing which can have ***multiple „states“*** (e.g. washing, drying)

which has multiple ***„functions“ to be run repeatedly*** (e.g., run, fill, empty)

and/or of which there may be ***multiple „instances“*** (machine 1, machine 2)

Object orientation: formal structure

Class **WashingMachine** 

Public Variables

WashingMode **IsRunning** **Contents**

variables which „outside“ may see and set

Private Variables

WaterFlow **Spin** **PowerConduit4State**

variables which are only internally used

Public Methods

Run(duration) **Fill(contents)**

functions which „outside“ may call/use

Private Methods

RegulateFlow(mode) **SpinUp(target)**

functions used internally e.g., by public methods

Constructor Method called when creating an instance of the class

---

Class is „blue-print“, actual „object“ is created via the constructor

MyWM = WashingMachine.create(model = „WashMaster 4000“)

Classes and Objects

***Class* WM** 

Public/Private Variables „property“ and „state“ descriptors

Public/Private Methods „function“ and „interface“ points

Constructor Method called when creating an instance of the class

---

Class is „blue-print“, actual „object“ is created via the constructor

MyWM = WashingMachine.create(model = „WashMaster 4000“)

„**MyWM** is an object of class **WM**“ „Object **MyWM** inhabits class **WM**“

Many languages distinguish class and object/instance variables/methods:

Object/instance variable and method values may differ by instance

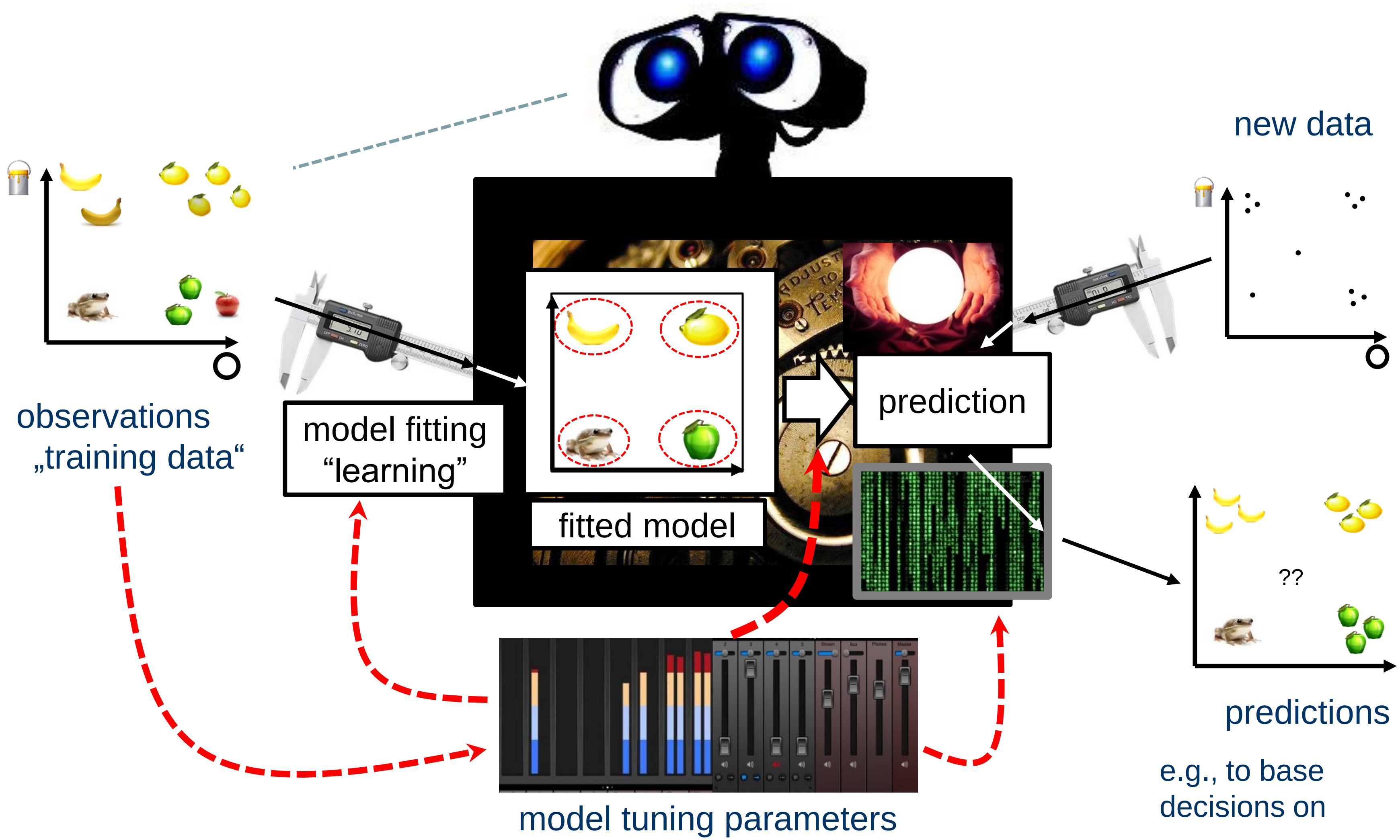
Class variable values are equal for all instances (often constant)

Class methods operate on and access only class variables

Usually: Class variable = „property“ Instance variable = „state“

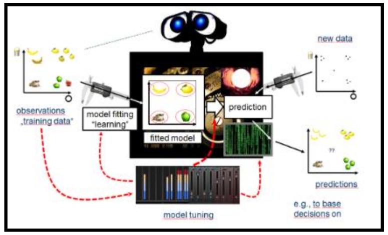
ModelType **PowerOutletType** **IsRunning** **Spin**

Learning Machines as Classes/Objects?



Examples: generalized linear model, linear regression, support vector machine, neural networks **(= *„deep learning“*),** random forests, gradient boosting,

The case for OO in stats/ML



When to use object orientation:

to model a thing which can have ***multiple „states"*** (e.g. washing, drying)

which has multiple ***„functions" to be run repeatedly*** (e.g., run, fill, empty)

and/or of which there may be ***multiple „instances"*** (machine 1, machine 2)

Learning strategies have ***multiple „states" (variables):***

fitted vs unitialized (private) hyperparameter settings (public)

Learning stratgies have ***methods to be run repeatedly:***

fitting to (new, more) data predicting on (train/test) data

Learning strategies arise in ***multiple instances:***

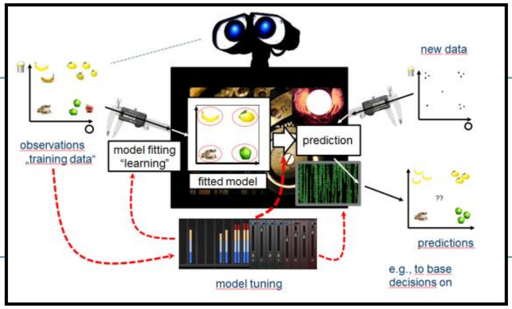
OLS vs neural network SVM1 (Gauss), SVM2 (polynomial)

All supervised learning strategies look the same from outside!

(except through meta-data, different hyper-parameter sets, and parameter settings)

Supervised learners as classes (mlr, sklearn)

***Class* SupervisedLearner**



Public Class Variables metadata (e.g., name, task)

Hyper-parameter *dictionary* (type/range & defaults)

Public Instance Variables

Hyper-parameter *settings* (values) : paramset

Private Instance Variables Fitted model : model

(formal types of these: fixed or dynamic)

***Public Method* Fit:** df$(N \times n)$ × df$(N \times 1)$ × paramset → model

„fit model to data“ external/public inputs (public var) (private var)

***Public Method Predict*:** df$(M \times n)$ × model × paramset → df$(M \times 1)$

„predict on new data“ external (private var) (public var) public output

***Public Method Fit&Predict*:** (optional – concatenation of fit and predict)

$$\text{df}(N \times n) \times \text{df}(N \times 1) \times \text{df}(M \times n) \times \text{paramset} \to \text{df}(M \times 1)$$

Constructor: instantiate hyper-parameters (sensible defaults), null model

OO: Inheritance and Polymorphism

Problem: *which models does the class model model?*

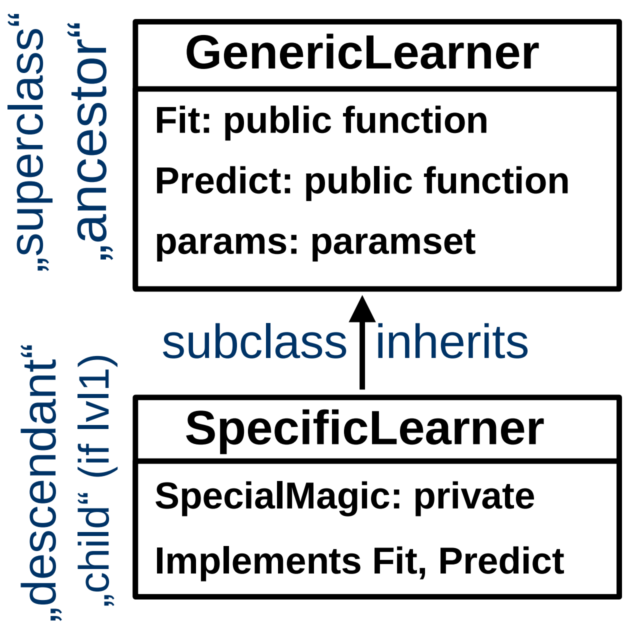
i.e., is the class a specific model, e.g., SVM or neural network?

which may need specific private methods such as backprop?

or a generic supervised learning model?

in which case model class specific routines are crammed into fit etc?

Solution: class inheritance



SpecificLearner inherits all variables/methods

i.e., call of **SpecificLearner.fit(x,y)**

defaults to **GenericLearner.fit(x,y)**

unless **SpecificLearner.fit(x,y)** specified

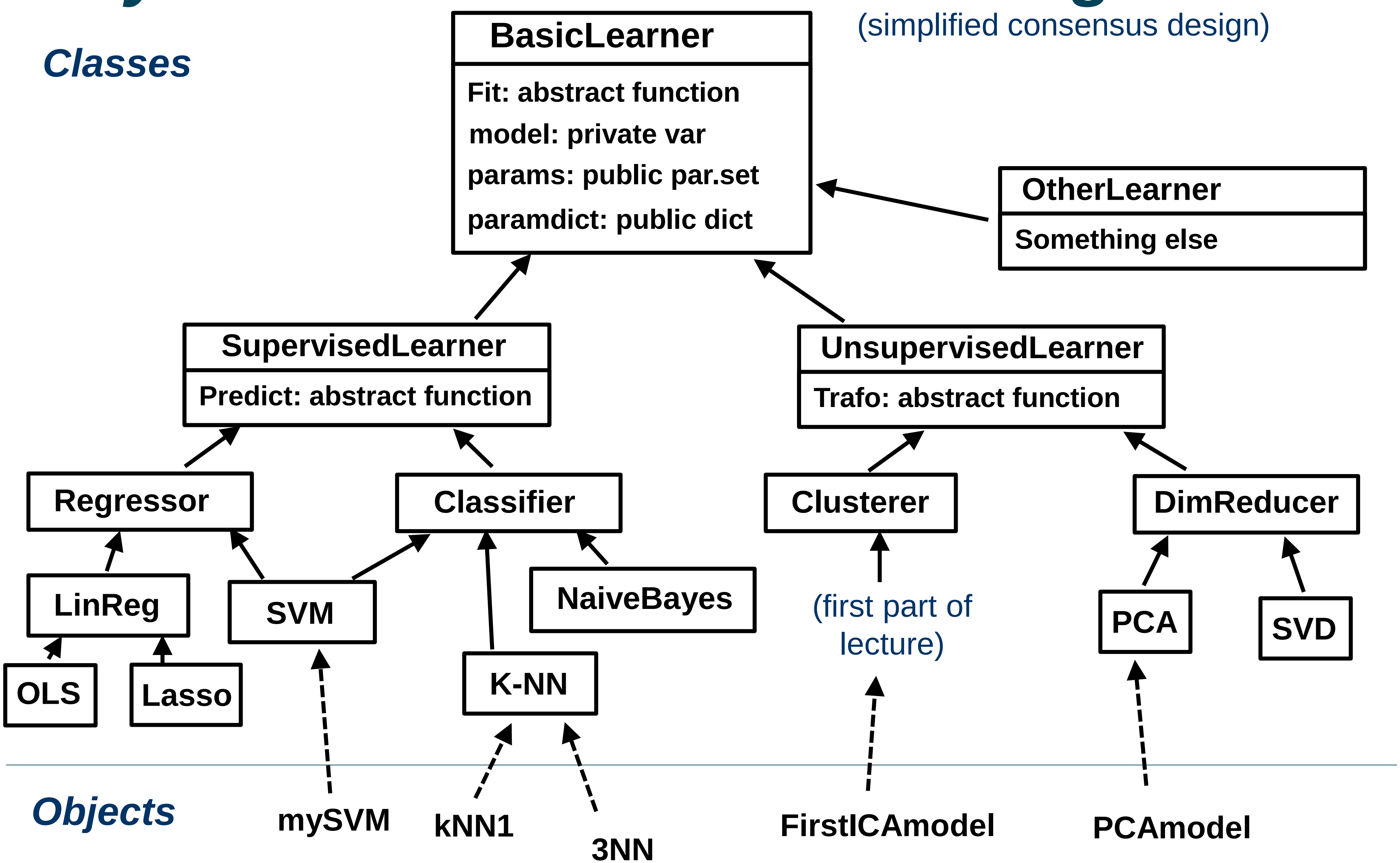
Subtype polymorphism:

sub-class instances behave like ancestor class

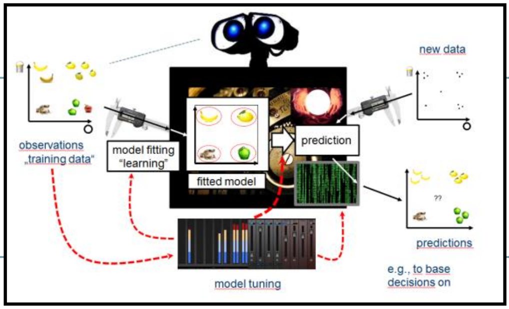
Abstract (base) class: specifies field

only descendants need to implement methods

Stylized model inheritance diagram



Unsupervised learner class API



Class **UnsupervisedLearner**

Public Class Variables metadata (e.g., name, task)

Hyper-parameter *dictionary* (type/range & defaults)

Public Instance Variables

Hyper-parameter *settings* (values) : paramset

Private Instance Variables Fitted model : model

(formal types of these: fixed or dynamic)

***Public Method* Fit:** df$(N \times n) \times$ paramset $\rightarrow$ model

„fit model to data“ external (public var) (private var)

No labels!

***Public Method Trafo*:** df$(M \times n) \times$ model $\times$ paramset $\rightarrow$ df$(M \times k)$

„produce features“ external (private var) (public var) public output

***Public Method Fit&Trafo*:** (optional – concatenation of fit and predict)

df$(N \times n) \times$ df$(M \times n) \times$ paramset $\rightarrow$ df$(M \times k)$

Constructor: instantiate hyper-parameters (sensible defaults), null model

Class types and ML output types

What distinguishes regression vs classification sub-class?

Class **SupervisedLearner**

***Public Method* Fit:** df$(N \times n) \times$ df$(N \times 1) \times$ paramset $\rightarrow$ model

***Public Method Predict*:** df$(M \times n) \times$ model $\times$ paramset $\rightarrow$ df$(M \times 1)$

In math: predict : $\mathcal{X}^M \times$ model $\times$ paramset $\rightarrow \mathcal{Y}^M$

If $\mathcal{Y}$ is a discrete set: *„classification"* If $\mathcal{Y} = \mathbb{R}$, then *„regression"*

Class **UnsupervisedLearner**

***Public Method* Fit:** df$(N \times n) \times$ paramset $\rightarrow$ model

***Public Method Trafo*:** df$(M \times n) \times$ model $\times$ paramset $\rightarrow$ df$(M \times k)$

In math: trafo : $\mathcal{X}^M \times$ model $\times$ paramset $\rightarrow \mathcal{Y}^M$

If $\mathcal{Y}$ is a discrete set: *„clustering"* If $\mathcal{Y} = \mathbb{R}$, then *„dimension reduction"*

Both cases: *„feature extraction"*, *„variable transformation"*

Implementation details: mlr, sklearn

mlr uses Bernd Bischl's custom OO functionality from BBmisc package

(not R's S3 or S4 OO framework)

„instance" = constructed by makeLearner

fit/predict/params and fit/trafo/params API

metadata for interaction for other classes/modules (see later)

no formal distinction between public/private variables/methods

no inheritance, sub-classes, or automatic sub-class polynorphism

---

sklearn uses python classes with task-speiic mixin (RegressorMixin etc)

„instance" = constructed by __init__

fit/predict/params and fit/trafo/params API

no metadata; hyper-parameter dictionary via get_params

(only partial distinction between „dictionary" and „setting")

no formal distinction between public/private variables/methods

Composition, Interaction & Workflow API

Key requirements to an ML toolbox

Model interface: provide access to a wide class of models

e.g., OLS, support vector machines, neural networks, etc

Fitting and prediction: simple interface given train/test data

Settable hyper-parameters: easily accessible and changeable

This should be similar for all classes and kinds of models

Model tuning & composition: grid-tuning, ensembling, pipelines

Specification of tuning/composition parameter same for all models

Exposing meta-model parameters as hyper-parameters of result

Model validation & evaluation: estimation of generalization loss

for standard loss metrics and re-sampling based validation schemes

Running of benchmarking experiments including all the above

User/workflow interaction: experiment set-up and reporting

All enablers of reproducibility and scientific transparency!

Important OO API design patterns for ML

Following nomenclature of „Design Patterns“ aka „Gang of Four“ (1994)

(see there for more patterns)

Composite pattern (structural):

Combines multiple objects to one

Model selectin, ensembles and pipelines (transformers plus learners)

Wrapper/adaptor pattern (structural):

Wraps around an object providing a new interface

Hyper-parameter tuning, task reduction, transformation pipelines

Facade/provider pattern (structural):

Provides a consistent, simple API towards a complex system

Interfacing of existing strategy/algorithm libraries, or data sources

Patterns for benchmarking and experimentation:

Command (orchestrate objects) and observer (report objects) patterns

Line up methods/strategies for benchmarking, collect/present results

The hyper-parameter tuning wrapper

tune:

SupervisedLearner
Fit: public function
Predict: public function
params: paramset

$\times$ tunectrl $\rightarrow$

SupervisedLearner
Fit: public function
Predict: public function
params: paramset

class types

$$(\text{untunedlearner} \times \text{tuneparams}) \mapsto \text{tunedlearner}$$

objects

„tunectrl“ is usually an object or structured list with tuning instructions:

which meta-strategy is used for tuning (grid search?)

Which hyper-parameters of the wrapped strategy are tuned how

Hyper-parameters of the meta-strategy

Tuning *removes* tuned parameters as externally/publicly accessible

adds hyper-parameters of the meta-strategy to the interface

$$\text{tunedlearner.params} = \text{untunedlearner.params} \setminus \text{tuneparams.tuned} \cup \text{tuneparams.ctrl}$$

The hyper-parameter tuning wrapper

tune: $\mathrm{SupervisedLearner} \times \mathrm{tunectrl} \to \mathrm{SupervisedLearner}$

$$(\mathrm{untunedlearner} \times \mathrm{tuneparams}) \mapsto \mathrm{tunedlearner}$$

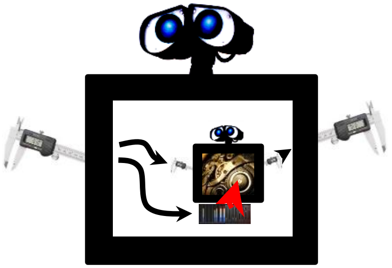
Important: the data is not an input of tune or part of tuneparams!

Why:
```
tunedlearner.fit(data) = {
    for params in tuneparams.paramgrid
        untunedlearner.params <- params
        loss[params] <- benchmark(data,untunedlearner)
    untunedlearner.params <- argmin(perf) }

tunedlearner.predict(data) = untunedlearner.predict(data)
```

The data argument is only implicitly used!

Data is passed only in the fitting/training phase.

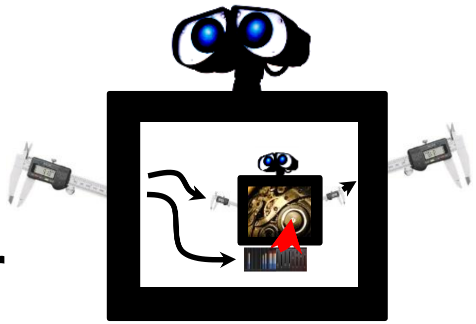


The hyper-parameter tuning wrapper

```
tunedlearner.fit(data) = {
    for params in tuneparams.paramgrid
        untunedlearner.params <- params
        loss[params] <- benchmark(data,untunedlearner)
    untunedlearner.params <- argmin(perf) }

tunedlearner.predict(data) = untunedlearner.predict(data)
```

Why is this a wrapper?



In the object oriented paradigm:

It is „natural“ to have **untunedlearner**

as a private (learner-valued) variable of **tunedlearner**

„ **tunedlearner** wraps around **untunedlearner** “

Composition and pipelining

pipeline: $\mathsf{UnsupLearner} \times \mathsf{SupLearner} \times \mathsf{pipectrl} \rightarrow \mathsf{SupLearner}$

(simplest case) $(\mathsf{unsupL} \times \mathsf{supL} \times \mathsf{pipeparams}) \mapsto \mathsf{pipeline}$

„first do feature extraction, then prediction on the full feature set“

pipeline.fit(traindata) = {

unsupL.fit(traindata)

supL.fit(unsupL.trafo(traindata)) }

pipeline.predict(testdata) = {

supL.predict(unsupL.trafo(testdata)) }

Hyper-parameters of **pipeline**: union of **(pipectrl, supL, unsupL).params**

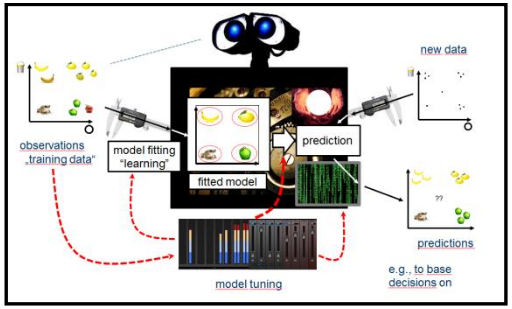
supL, unsupL are **encapsulated,** private variables of **pipeline**

Generalization: directed graphical modelling flow („feature split/union“)

The full object-oriented ML Toolbox API

Encapsulation points as found in the **R/mlr** or **scikit-learn** packages

´within the statistical/ML modelling workflow



modular structure

object orientation

„learning machine“ object

Linear regression

fit(traindata)

predict(testdata)

plus metadata & model info

Abstraction models objects with unified API:

Concept abstracted	Public interface	in **R/mlr**	in **sklearn**
Learning Machines	fitting, predicting, set parameters	**Learner**	**estimator**
Re-sampling schemes	sample, apply & get results	**ResampleDesc**	splitter classes in **model_selection**
Evaluation metrics	compute from results, tabulate	**Measure**	metrics classes in **metrics**
Meta-modelling Tuning Ensembling Pipelining	wrapping machines by strategy	various wrappers fused classes	various wrappers **Pipeline**
Learning task	benchmark, list strategies/measures	**Task**	Implicit, not encapsulated

Further API interface points

Performance/loss evaulators: e.g., MAE, RMSE, sensitivity

Encapsulate loss functions and generic scoring/utility rules

Needs to know: mean loss or not? How to obtain StdErr/CI?

Re-sampling schemes: e.g., „k-fold CV“ or „bootstrap“

Inherent distinction between „instructions“ and „concrete splits“

Closely related to „iterator pattern“ and „bridge pattern“

Modelling tasks: e.g., „predict variable Y in dataset X“

Bundle dataset with the task to perform on it, usually pointer type

„bridge/proxy pattern“ for cross-linking and provision to orchestrator

Workflow abstraction: e.g., „instructions for analyses in paper X“

Full instructions for conducting analyses, e.g., benchmarking experiment

„command pattern“ for execution by an orchestrator or mediator

WORQ - widely open research questions!

(aka BSc/MSc/PhD topics for the coding and/or practically inclined)

Full workflow integration and orchestration API design

Running of full benchmarking experiments is not fully supported

(or not „cleanly“ supported, e.g., the mlr design is nor properly OO)

Toolboxes implement mostly prediction, *inference* poorly supported

Data provider and high-performance computing back-ends

Clean toolbox designs assume that the data fits into workspace

CPU/GPU computing and distributed job scheduling is a challenge

Existing solutions are in early stages, despite high demand & relevance

Basic toolbox design for the (unsolved!) „complicated tasks“

Time series, on-line learning, anomaly detection, reinforcement learning

Structured and heterogeneous prediction tasks

Probabilistic and Bayesian modelling

Meta-data structure and smart automated modelling

Machine learning for selecting the best machine learning approach