Scaling up complexity

Big data tools for Physics and Astronomy, CWI Amsterdam

Zahari Kassabov

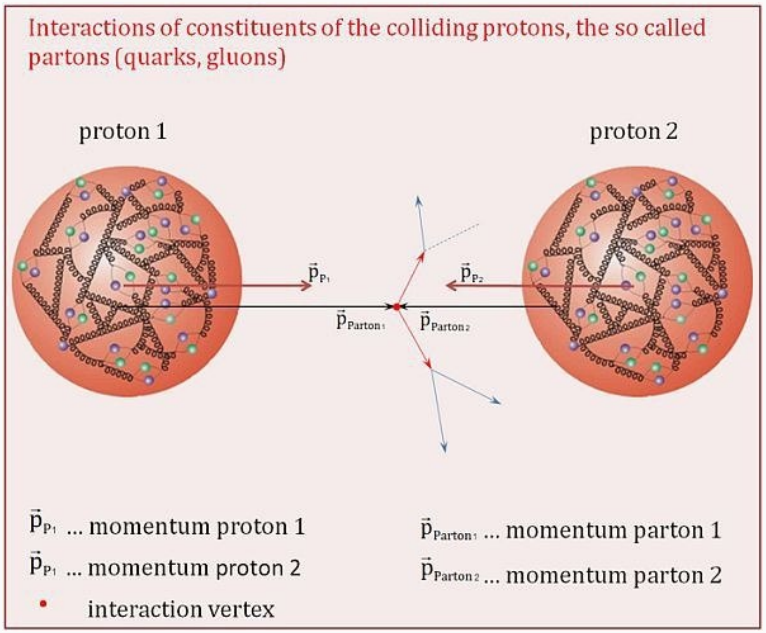
June 19th 2017

University of Turin, University of Milan

UNIVERSITA
DEGLI STUDI
DI TORINO

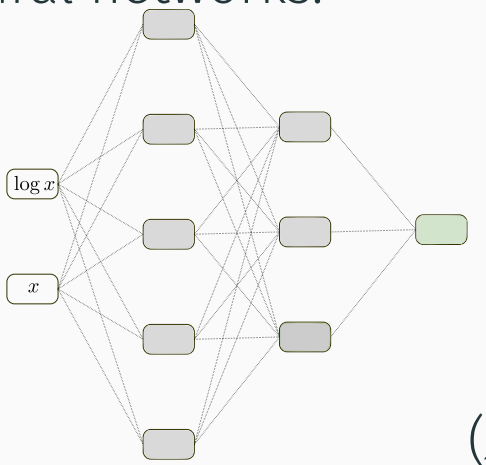
NNPDF in two slides

- I work in the NNPDF collaboration.
- We provide predictions for LHC processes. Specifically, we determine the *Parton Distribution functions* of the proton.
- Think of it as the probability density of sampling given quark or a gluon with a momentum p_{parton_1} (+quantum corrections).



NNPDF innovations related to statistical learning[1]

- Do not assume an overly restricted model for the PDFs. Paramterize them with neural networks.



$$pdf(x, Q_0) = Cx^{-\alpha}(1-x)^{\beta} \times \qquad (x = p_{\text{parton1}}/p_{p1})$$

- A method to propagate uncertainty on the experimental inputs: Psudorreplica sampling.
- A method to validate the methodology: Closure testing.

[1] See talk on IML workshop last Friday

Motivating example: NNPDF 3.1

- NNPDF 3.0 released at the end of 2014.
- NNPDF 3.1 (released last month arxiv:1706.00428) was planned as a small update:
  - Add a bunch of new data (mostly for LHC experiments)
  - Improvement in methodology (fitting the charm quark PDF).
- How difficult can it be?
  - *A lot*![2] LHC data provide very precise constraints.
  - At this precision a lot of previously neglected effects become important (think interpolation accuracy, Monte Carlo integration).
- Problem not fundamentally different but requires more complex scrutiny.
- Need to test compare and understand precisely all our inputs
  $\longrightarrow$ Need powerful analysis code.

[2] See talk at HiggsTools Turin last month

This talk

Alternative title *“How I wanted to make a plotting tool and ended up building a compiler”*

What we want

- Flexible analysis framework.

What we want

- Flexible analysis framework.
  - *Plot X as a function of Y without many assumptions on what they are.*

What we want

- Flexible analysis framework.
  - *Plot X as a function of Y without many assumptions on what they are.*
- High level declarative input: *What you want, not how.*

What we want

- Flexible analysis framework.
  - *Plot X as a function of Y without many assumptions on what they are.*
- High level declarative input: *What you want, not how.*
- Initialization stage: Check all the inputs

What we want

- Flexible analysis framework.
  - *Plot X as a function of Y without many assumptions on what they are.*
- High level declarative input: *What you want, not how.*
- Initialization stage: Check all the inputs
- Results as reproducible and backwards compatible as possible.

What we want

- Flexible analysis framework.
  - *Plot X as a function of Y without many assumptions on what they are.*
- High level declarative input: *What you want, not how.*
- Initialization stage: Check all the inputs
- Results as reproducible and backwards compatible as possible.
- Loops are important. Make them first class.

Reportengine

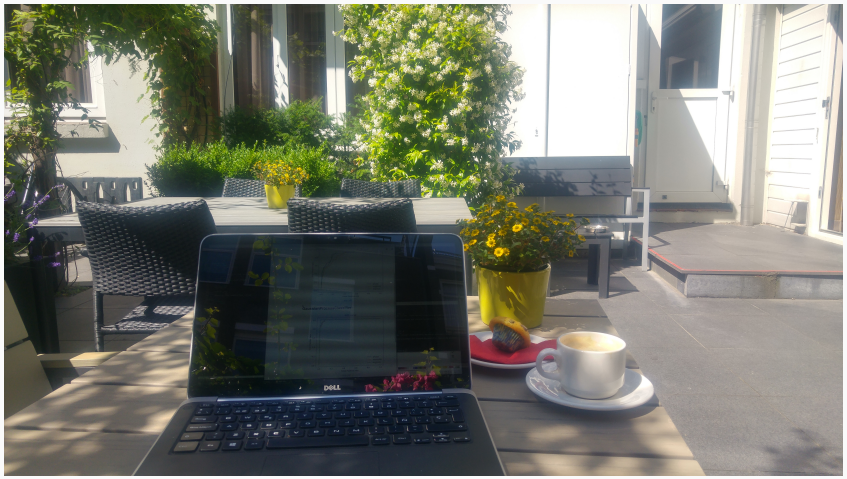
https://github.com/NNPDF/reportengine

A *framework* for scientific code.

- Advanced configuration parser.
- Call graph generator and executor.
- Supports for contracts.
- General command support (e.g. colors, error handlers).
- A report application built on top of it.
- How difficult can be?

The Iris flower dataset

- 150 samples from 3 species of *Iris flowers*, from a single pasture in the Gaspé Peninsula.
- Measured 4 features: Sepal width and height, petal width and height.
- Popularized by Fisher, in works on Linear Discriminant analysis.



Example application: flowers

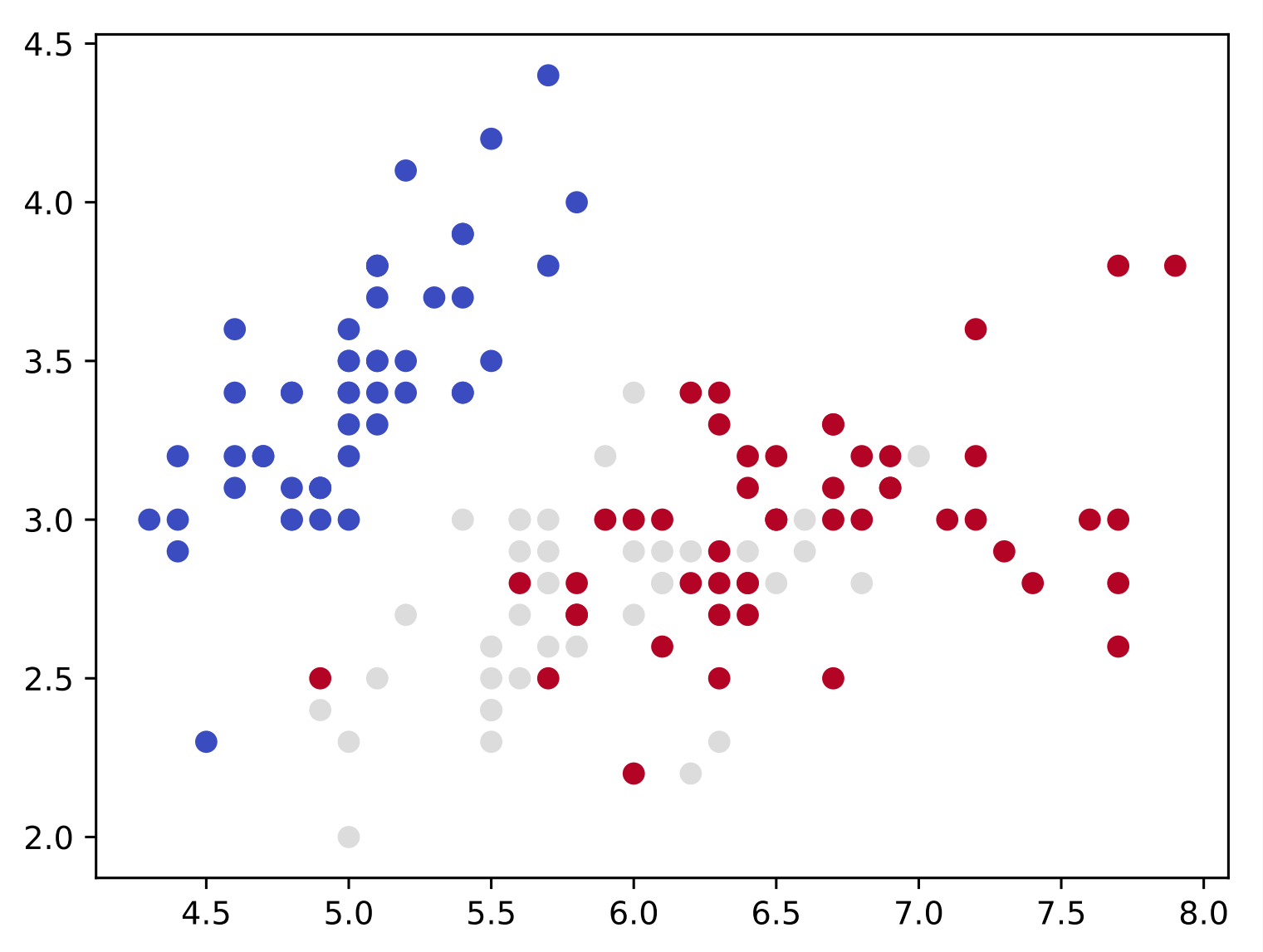
- An example app to study the IRIS dataset.
- Objectives:
  - Demonstrate design requirements and main features of the framework.
  - Examples that fit on a slide.
- Non objectives:
  - Usefulness.
  - Nice plots.
- Using scikits-learn because it has a nice API.

Simple input

```
#input.yaml
xaxis: "sepal length (cm)"
yaxis: "sepal width (cm)"

actions_:
    - - plot_2d
```

Simple output



Let’s implement it all!

A little boilerplate

```
#flowers/app.py An IRIS study application
from reportengine.app import App
from reportengine.report import Config
from reportengine.configparser import ConfigError
class FlowersConfig(Config):
    ...
class FlowersApp(App):
    config_class = FlowersConfig
def main():
    a = FlowersApp(name="Flowers",
        default_providers=['flowers.actions',
        'reportengine.report'])
    a.main()

if __name__ == '__main__':
    main()
```

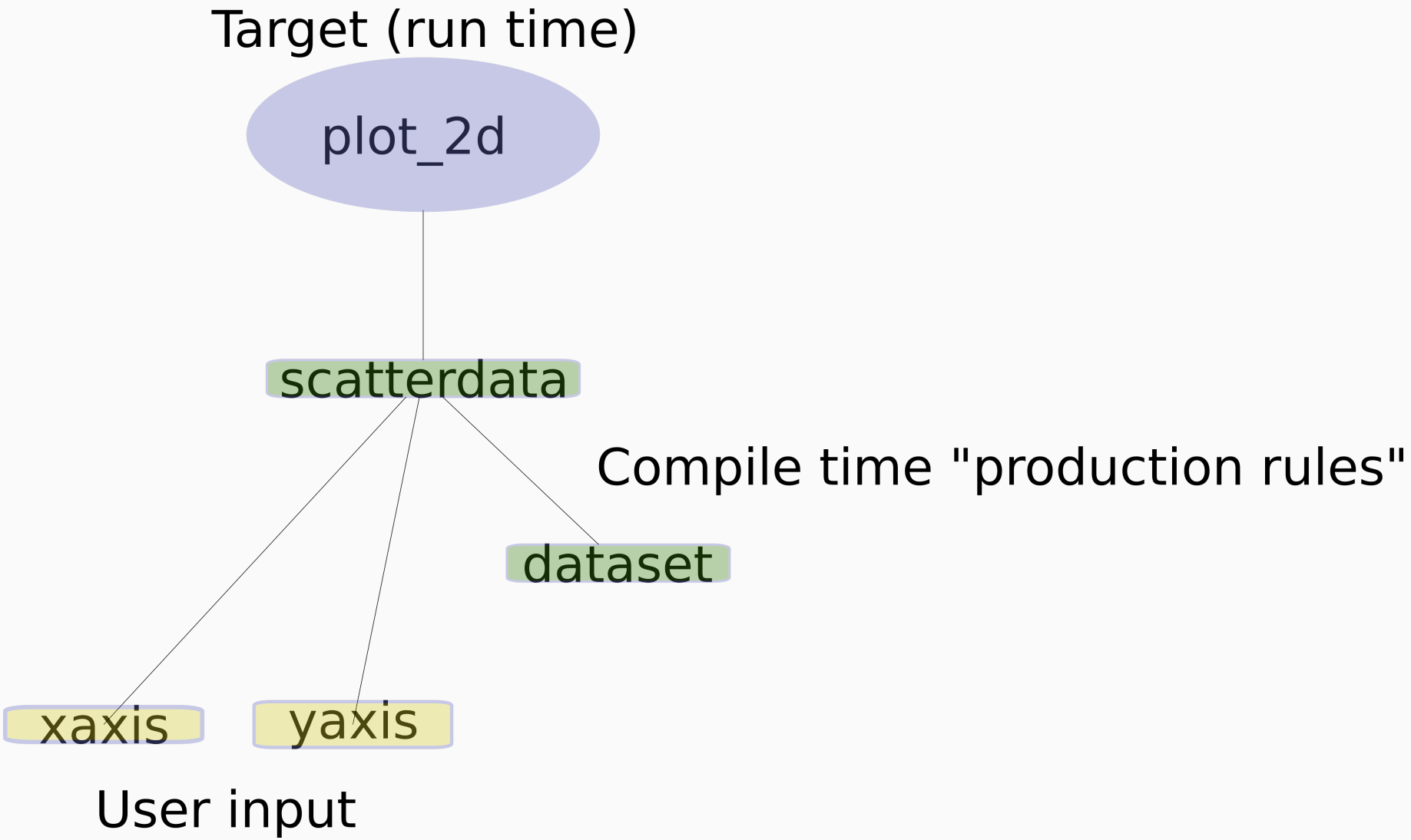
Basic API elements

- Config class
  - “Compile time” functionality. Parses inputs and generates resources before the slow functions run.
  - Two types of functions:

    Input parsers “parse_<KEY>” Take KEY as argument from the input card and produce a result.

    Production rules “produce_<KEY>” take zero or more inputs and produce the resource KEY based on them.
- Provider modules
  - Just normal python modules!
  - Containing functions executed at runtime.

Compilation and execution plan



Argument binding by name

- In reportengine *f*(*argname*), means that to execute *f* we must find a resource called *argname*
- *argname* Could be
  - An input parameter
  - A production rule
  - A runtime function

Config

```
class FlowersConfig(Config):
    def produce_dataset(self):
        """Produce the IRIS dataset"""
        return sklearn.datasets.load_iris()
    def parse_xaxis(self, feature:str):
        """Feature in the X axis"""
        return feature
    def parse_yaxis(self, feature:str):
        """Feature in the Y axis"""
        return feature
    def produce_scatterdata(self, dataset, xaxis, yaxis):
        """Check that features exist and return the data values"""
        def ind(name):
            try:
                 return dataset.feature_names.index(name)
            except ValueError as e:
                raise ConfigError(f'No such feature', bad_item=name,
                    alternatives=dataset.feature_names) from e
        i,j = ind(xaxis), ind(yaxis)
        return dataset.data[:, i], dataset.data[:,j], dataset.target
```

Provider

```
#flowers/actions.py Tools to analyze the IRIS dataset
from reportengine.figure import figure
@figure
def plot_2d(scatterdata):
    """Generate scatter plot of the values of xaxis vs yaxis"""
    x,y,category = scatterdata
    fig, ax = plt.subplots()
    ax.scatter(x,y, c=category, cmap=plt.cm.coolwarm)
    return fig
```

Done!

```
$ flowers input.yaml  -o output
[INFO]: All requirements processed and checked successfully.
Executing actions.
$ ls output/figures/
plot_2d.pdf  plot_2d.png
```

Much more than giving the correct result: Errors

```
$ cat typo.yaml
xaxis: "sepal lenght (cm)"
yaxis: "sepal width (cm)"
actions_:
    - - plot_2d
$ flowers typo.yaml
[ERROR]: Bad configuration encountered:
No such feature
Instead of 'sepal lenght (cm)', did you mean one of the following?
 - sepal length (cm)
 - petal length (cm)
 - sepal width (cm)

$ head -1 badtype.yaml
xaxis: 43
$ flowers badtype.yaml
[ERROR]: Bad configuration encountered:
Bad input type for parameter 'xaxis': Value '43' is not of type
str, but of type 'int'.
```

Much more than giving the correct result: Help

```
$ flowers --help plot_2d
Defined in: flowers.actions
Generates: figure
plot_2d(scatterdata)
Generate scatter plot of the values of xaxis vs yaxis
The following resources are read from the configuration:
    xaxis(str): Feature in the X axis   [Used by scatterdata]
    yaxis(str): Feature in the Y axis   [Used by scatterdata]
```

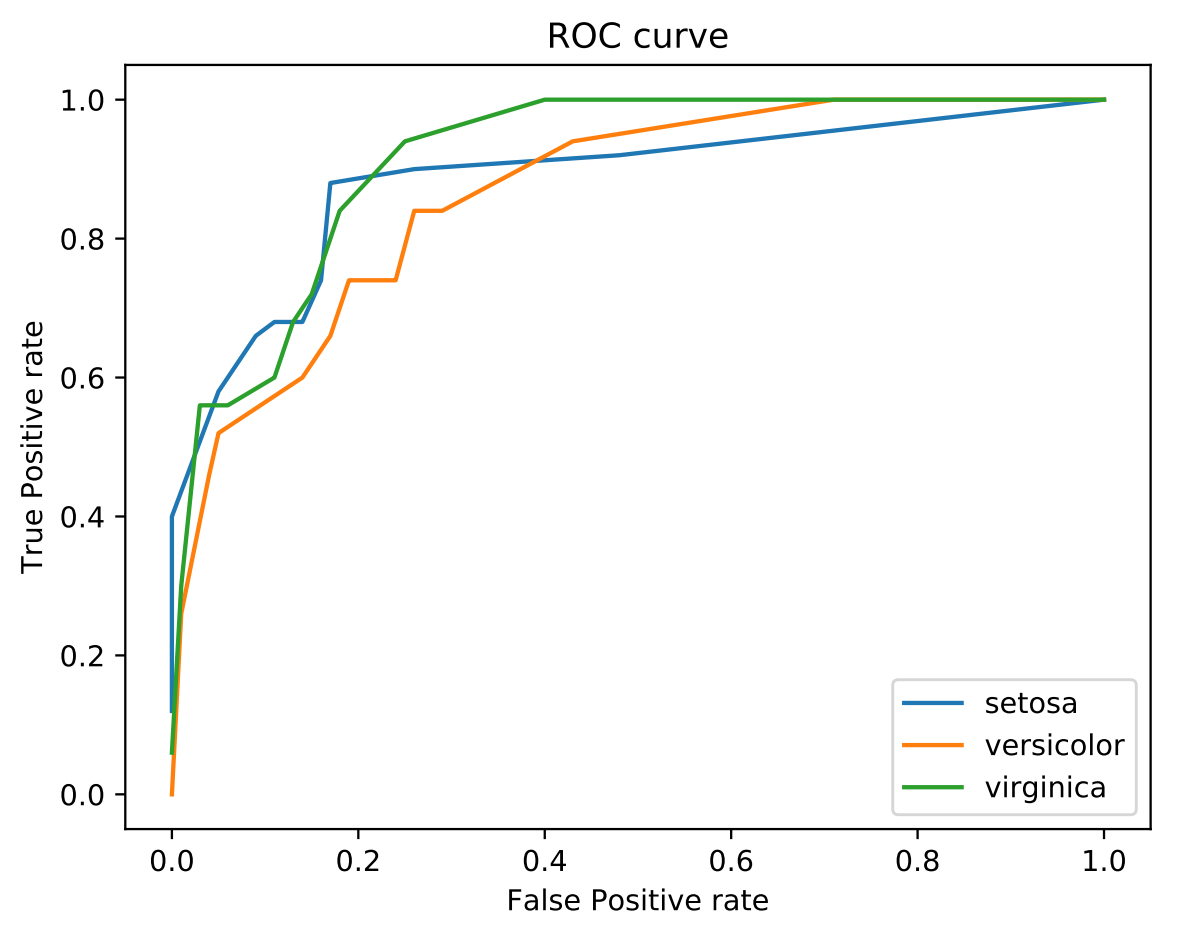
```
#app.py
ALGORITHMS = {
    'svc': sklearn.svm.SVC,
    'gp': sklearn.gaussian_process.GaussianProcessClassifier,
    'random_forest': sklearn.ensemble.RandomForestClassifier
}
#class Config
    def parse_algorithm(self, alg:str):
        """The name of the classification algorithms"""
        ...

#actions.py
def fit_result(algorithm, dataset):
    """Fit to a sample of the dataset where some labels have been
    shuffled  using the default algorithm
    parameters"""
    ...

@figure
def plot_roc(fit_result, algorithm, dataset):
    ...
```

Random Forest ROC curve

```
$ cat input.yaml
algorithm: random_forest
actions_:
    - - plot_roc
$ flowers input.yaml
```



Support vector: Crash!

```
$ cat input.yaml
algorithm: svc
actions_:
    - - plot_roc
$ flowers input.yaml
[INFO]: All requirements processed and checked successfully.
Executing actions.
----
Traceback (most recent call last):
...
AttributeError: predict_proba is not available when
probability=False
----
[CRITICAL]: A critical error oucurred. It has been logged in
/tmp/Flowers-crash-svp6cymq
```

What happened

- Support Vector Machines is (by default) a categorical classifier so the ROC curve doesn't make sense.
- Fitting the data itself makes perfect sense, but the ROC action should require probabilistic algorithms.
    - Let's assume we don't want to use the probabilistic version of SVC.
- We should require it at initialization time, not after the expensive fitting.

Contracts

- Executed at initialization time.
- Can execute arbitrary logic on whatever is known at that moment.

```
from reportengine.checks import make_argcheck, CheckError
@make_argcheck
def _check_can_predict_probabilities(algorithm):
    res = hasattr(algorithm().fit([[0],[1]], [0,1]), 'predict_proba')
    if not res:
        raise CheckError(f"Algorithm {algorithm.__name__}"
            "doesn't support 'predict_proba'")

@figure
@_check_can_predict_probabilities
def plot_roc(fit_result, algorithm, dataset):
    ...
```

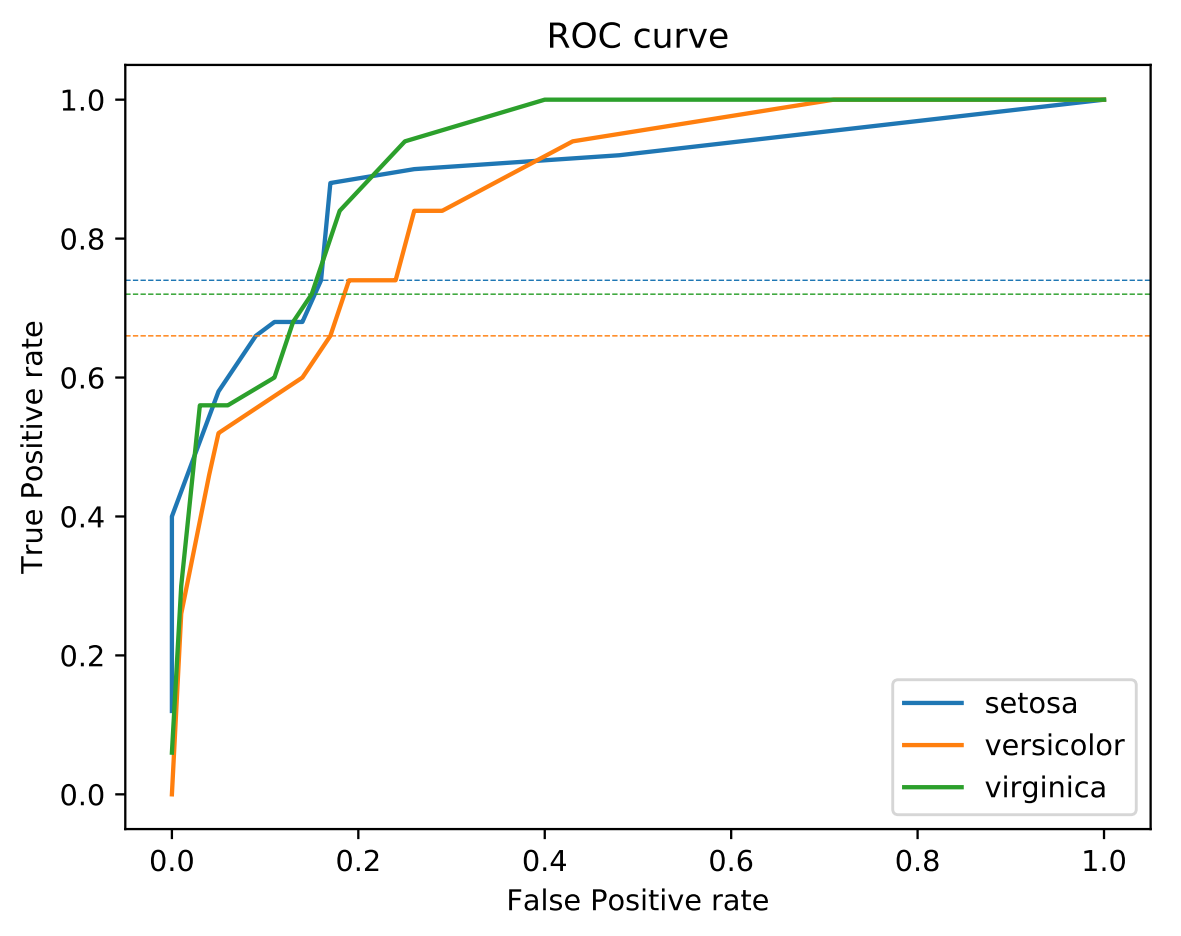
Parameters

- Read from input directly.
- Automatically type checked.
- Can have defaults, and further checks.

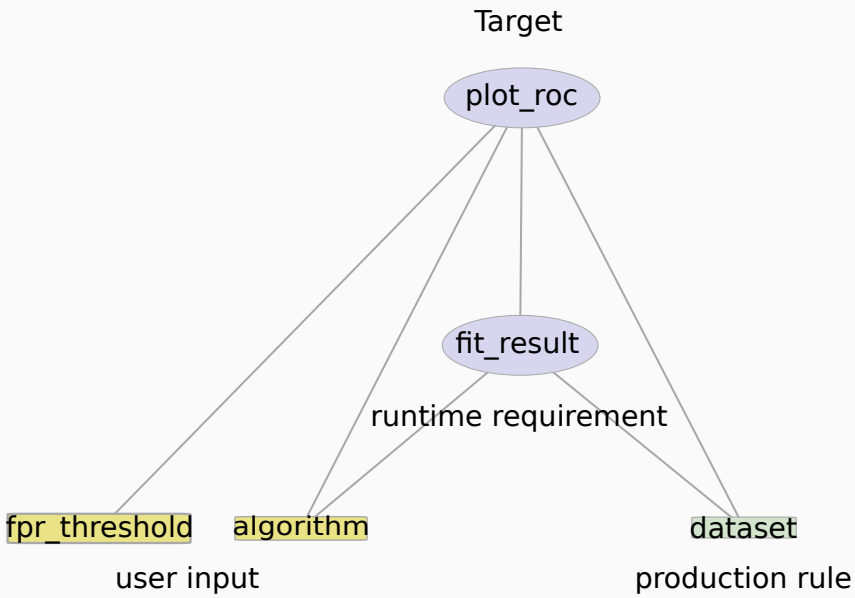
```
@make_argcheck
def _check_fpr_threshold(fpr_threshold):
    """Check that it's in (0,1) if given"""
    ...
@figure
@_check_fpr_threshold
@_check_can_predict_probabilities
def plot_roc(fit_result, algorithm, dataset,
        fpr_threshold:(float, type(None))=None):
    """Plot the ROC curve for each category. Mark the true positive
    rate at the ``fpr_threshold`` if given"""
    ...
```

Plot with parameter

```
#input.yaml
fpr_threshold: 0.15
algorithm: random_forest
actions_:
    - - plot_roc
```



Graph building and execution



Roughly when a runtime node is required:

1. Find requirements.
2. Verify requirements recursively.
3. Execute checks (contracts).
4. Requirement is satisfied.

Execute runtime nodes (in dependency order) when all *targets* (user requested actions) have been verified.

Loops and reports. But which algorithm is better?

- We would like to find out whether Gaussian Processes or Random Forests produce a better ROC curve.
  - We want to compute it for each.
  - Need a way to express loops (and in general namespaces) in a declarative way.
- We would like to see the results organized together.

Report application

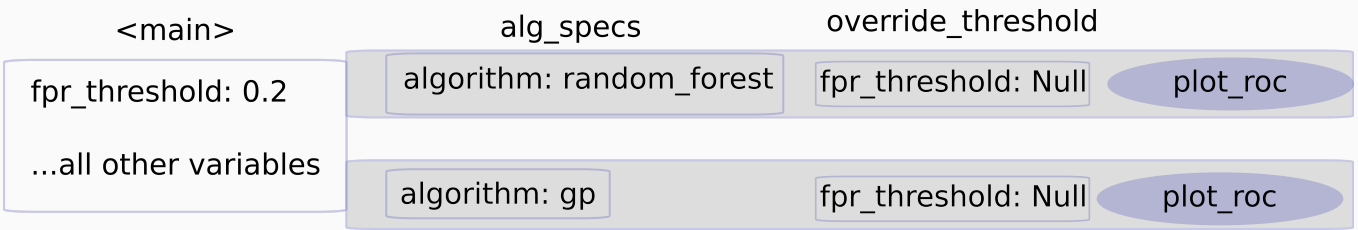
- Special tags {@@} are evaluated and used to infer *targets*.
- Targets are validated and then executed, just as described.
- Text is substituted, and the resulting *markdown* is passed to PANDOC to generate HTML.

An example

```
fpr_threshold: 0.2
#any name
override_threshold:
    fpr_threshold: Null
#any name
alg_specs:
    - algorithm: random_forest
    - algorithm: gp
template_text: |
    % IRIS dataset with some shuffled labels
    % Zahari Kassabov
    {@with alg_specs::override_threshold@}
    {@algorithm@}
    ------------
    {@plot_roc@}
    {@endwith@}
actions_:
    - - report:
            main: True
```

Stack frame aggregation

- Associate a list of elements to each target:
  - This is what “{@with ...@}” does. E.g. plot_roc is “associated” to [’alg_specs’, ’override_threshold’]
- If an element:
  - is a mapping, push its variables to the stack
  - is a list of mappings, do the Cartesian product and execute the action for each possible stack.
- Repeat the text between {@with ...@} and {@endwith@} for each possibility.



Output

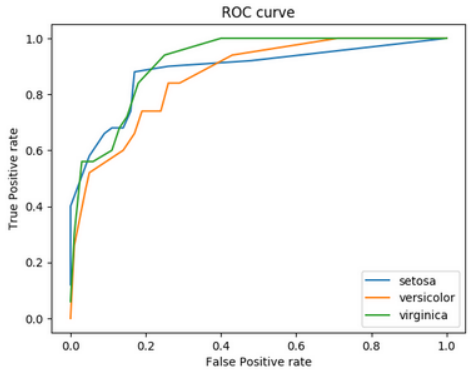
file:///home/zah/phd/reportengine/example/output/index.html 67% Search

IRIS dataset with some shuffled labels

Zahari Kassabov

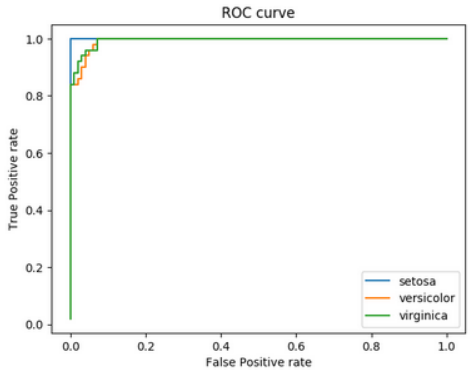
- RandomForestClassifier
- GaussianProcessClassifier

RandomForestClassifier



.png .pdf

GaussianProcessClassifier



.png .pdf

Powered by reportengine

Automatic lists

- If we know how to parse X, we know how to parse a list of X!

```
from reportengine.configparser import element_of
#class Config
    @element_of('algorithms')
    def parse_algorithm(self, alg:str):
        """The name of the classification algorithms"""
        ...
```

- Can shorten “alg_specs” from before:

```
algorithms:
  - random_forest
  - gp
```

Putting things together

```
from reportengine import collect
fit_results = collect(fit_result, ['algorithms'])
```

“Execute fit_result for each namespace in [’algorithms’] and put the flattened result in a list”

```
import pandas as pd
from sklearn.metrics import hamming_loss
from reportenfine.table import table
@table
def hamming_loss_table(algorithms, fit_results, dataset):
    records = []
    for algorithm, res in zip(algorithms, fit_results):
        records.append({'algorithm': algorithm.__name__, "Hamming "
                "loss": hamming_loss(dataset.target,
                    res.predict(dataset.data))})
    return pd.DataFrame(records)
```

Simple input

```
#report.yaml
algorithms:
    - random_forest
    - gp

template_text: |
    % IRIS dataset with some shuffled labels
    % Zahari Kassabov
    Hamming loss
    ============
    {@hamming_loss_table@}
    ROC curves
    ==========
    {@with algorithms@}
    {@algorithm@}
    ------------
    {@plot_roc@}
    {@endwith@}

actions_:
    - - report:
            main: True
```

Complex output

file:///home/zah/phd/reportengine/example/output/index.html 90% Search

IRIS dataset with some shuffled labels — Zahari Kassabov

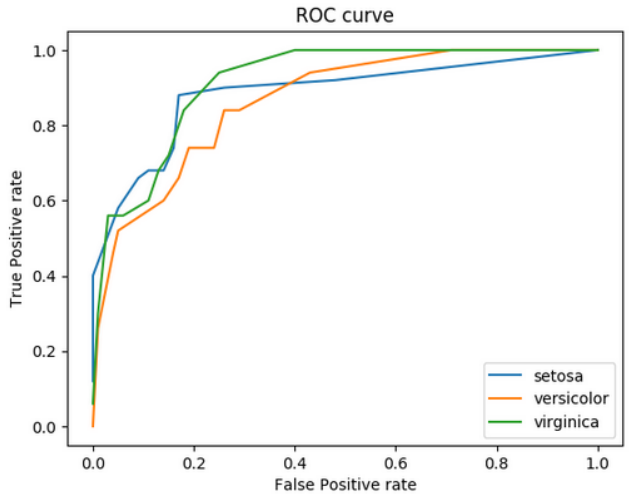
- Hamming loss
- ROC curves
  - RandomForestClassifier
  - GaussianProcessClassifier

Hamming loss

	Hamming loss	algorithm
0	0.313333	RandomForestClassifier
1	0.046667	GaussianProcessClassifier

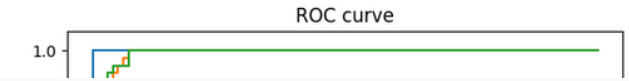
ROC curves

RandomForestClassifier



.png .pdf

GaussianProcessClassifier



Thank you!