pyhf Documentation

Release 0.1.1

Lukas Heinrich, Matthew Feickert

Dec 18, 2019

CONTENTS

Measurements in High Energy Physics (HEP) rely on determining the compatibility of observed collision events with theoretical predictions. The relationship between them is often formalised in a statistical *model* $f(x|)$ describing the probability of data x given model parameters . Given observed data, the *likelihood* $\mathcal{L}()$ then serves as the basis to test hypotheses on the parameters . For measurements based on binned data (*histograms*), the family of statistical models has been widely used in both Standard Model measurements [4] as well as searches for new physics [5]. In this package, a declarative, plain-text format for describing -based likelihoods is presented that is targeted for reinterpretation and long-term preservation in analysis data repositories such as HEPData [3].

CHAPTER

ONE

HISTFACTORY

Statistical models described using [2] center around the simultaneous measurement of disjoint binned distributions (*channels*) observed as event counts . For each channel, the overall expected event rate[1] is the sum over a number of physics processes (*samples*). The sample rates may be subject to parametrised variations, both to express the effect of *free parameters* [2] and to account for systematic uncertainties as a function of *constrained parameters* . The degree to which the latter can cause a deviation of the expected event rates from the nominal rates is limited by *constraint terms*. In a frequentist framework these constraint terms can be viewed as *auxiliary measurements* with additional global observable data , which paired with the channel data completes the observation $x = (,)$. In addition to the partition of the full parameter set into free and constrained parameters $= (,)$, a separate partition $= (,)$ will be useful in the context of hypothesis testing, where a subset of the parameters are declared *parameters of interest* and the remaining ones as *nuisance parameters* .

$$f(x|) = f(x|\overbrace{,}^{\text{free}}\underbrace{)}_{\text{constrained}} = f(x|\overbrace{,}^{\text{parameters of interest}}\underbrace{)}_{\text{nuisance parameters}} \tag{1.1}$$

Thus, the overall structure of a probability model is a product of the analysis-specific model term describing the measurements of the channels and the analysis-independent set of constraint terms:

$$f(,|,) = \underbrace{\prod_{c\in\text{ channels}}\prod_{b\in\text{ bins}_c}\text{Pois}\left(n_{cb}\,|\,\nu_{cb}\,(,)\right)}_{\substack{\text{Simultaneous measurement}\\ \text{of multiple channels}}}\qquad\underbrace{\prod_{\in} c(a|\,)}_{\substack{\text{constraint terms}\\ \text{for } x201C\text{auxiliary measurements}x201D}}\quad, \tag{1.2}$$

where within a certain integrated luminosity we observe n_{cb} events given the expected rate of events $\nu_{cb}(,)$ as a function of unconstrained parameters and constrained parameters . The latter has corresponding one-dimensional constraint terms $c_(a_|\,)$ with auxiliary data a constraining the parameter . The event rates ν_{cb} are defined as

$$\nu_{cb}\,() = \sum_{s\in\text{ samples}}\nu_{scb}\,(,) = \sum_{s\in\text{ samples}}\underbrace{\left(\prod_{\kappa\in\,\kappa}\kappa_{scb}\,(,)\right)}_{\text{multiplicative modifiers}}\underbrace{\left(\nu^0_{scb}\,(,) + \sum_{\Delta\in\Delta}\Delta_{scb}\,(,)\right)}_{\text{additive modifiers}}. \tag{1.3}$$

The total rates are the sum over sample rates ν_{csb}, each determined from a *nominal rate* ν^0_{scb} and a set of multiplicative and additive denoted *rate modifiers* $\kappa()$ and $\Delta()$. These modifiers are functions of (usually a single) model parameters. Starting from constant nominal rates, one can derive the per-bin event rate modification by iterating over all sample rate modifications as shown in (1.3).

As summarised in *Modifiers and Constraints*, rate modifications are defined in for bin b, sample s, channel c. Each modifier is represented by a parameter $\phi \in \{\gamma, \alpha, \lambda, \mu\}$. By convention bin-wise parameters are denoted with γ and

[1] Here rate refers to the number of events expected to be observed within a given data-taking interval defined through its integrated luminosity. It often appears as the input parameter to the Poisson distribution, hence the name "rate".

[2] These *free parameters* frequently include the of a given process, i.e. its cross-section normalised to a particular reference cross-section such as that expected from the Standard Model or a given BSM scenario.

interpolation parameters with α. The luminosity λ and scale factors μ affect all bins equally. For constrained modifiers, the implied constraint term is given as well as the necessary input data required to construct it. σ_b corresponds to the relative uncertainty of the event rate, whereas δ_b is the event rate uncertainty of the sample relative to the total event rate $\nu_b = \sum_s = \nu_{sb}^0$.

Modifiers implementing uncertainties are paired with a corresponding default constraint term on the parameter limiting the rate modification. The available modifiers may affect only the total number of expected events of a sample within a given channel, i.e. only change its normalisation, while holding the distribution of events across the bins of a channel, i.e. its "shape", invariant. Alternatively, modifiers may change the sample shapes. Here supports correlated an uncorrelated bin-by-bin shape modifications. In the former, a single nuisance parameter affects the expected sample rates within the bins of a given channel, while the latter introduces one nuisance parameter for each bin, each with their own constraint term. For the correlated shape and normalisation uncertainties, makes use of interpolating functions, f_p and g_p, constructed from a small number of evaluations of the expected rate at fixed values of the parameter α[3]. For the remaining modifiers, the parameter directly affects the rate.

Table 1: Modifiers and Constraints

Description	Modification	Constraint Term c	Input
Uncorrelated Shape	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \mathrm{Pois}\left(r_b = \sigma_b^{-2} \middle\vert \rho_b = \sigma_b^{-2}\gamma_b\right)$	σ_b
Correlated Shape	$\Delta_{scb}(\alpha) = f_p\left(\alpha \middle\vert \Delta_{scb,\alpha=-1}, \Delta_{scb,\alpha=1}\right)$	$\mathrm{Gaus}\left(a = 0 \middle\vert \alpha, \sigma = 1\right)$	$\Delta_{scb,\alpha=\pm1}$
Normalisation Unc.	$\kappa_{scb}(\alpha) = g_p\left(\alpha \middle\vert \kappa_{scb,\alpha=-1}, \kappa_{scb,\alpha=1}\right)$	$\mathrm{Gaus}\left(a = 0 \middle\vert \alpha, \sigma = 1\right)$	$\kappa_{scb,\alpha=\pm1}$
MC Stat. Uncertainty	$\kappa_{scb}(\gamma_b) = \gamma_b$	$\prod_b \mathrm{Gaus}\left(a_{\gamma_b} = 1 \middle\vert \gamma_b, \delta_b\right)$	$\delta_b^2 = \sum_s \delta_{sb}^2$
Luminosity	$\kappa_{scb}(\lambda) = \lambda$	$\mathrm{Gaus}\left(l = \lambda_0 \middle\vert \lambda, \sigma_\lambda\right)$	$\lambda_0, \sigma_\lambda$
Normalisation	$\kappa_{scb}(\mu_b) = \mu_b$		
Data-driven Shape	$\kappa_{scb}(\gamma_b) = \gamma_b$		

Given the likelihood $\mathcal{L}()$, constructed from observed data in all channels and the implied auxiliary data, *measurements* in the form of point and interval estimates can be defined. The majority of the parameters are *nuisance parameters* — parameters that are not the main target of the measurement but are necessary to correctly model the data. A small subset of the unconstrained parameters may be declared as *parameters of interest* for which measurements hypothesis tests are performed, e.g. profile likelihood methods [1]. The *Symbol Notation* table provides a summary of all the notation introduced in this documentation.

[3] This is usually constructed from the nominal rate and measurements of the event rate at $\alpha = \pm 1$, where the value of the modifier at $\alpha = \pm 1$ must be provided and the value at $\alpha = 0$ corresponds to the corresponding identity operation of the modifier, i.e. $f_p(\alpha = 0) = 0$ and $g_p(\alpha = 0) = 1$ for additive and multiplicative modifiers respectively. See Section 4.1 in [2].

Table 2: Symbol Notation

Symbol	Name
$f(x\|)$	model
$\mathcal{L}()$	likelihood
$x = \{,\}$	full dataset (including auxiliary data)
	channel data (or event counts)
	auxiliary data
$\nu()$	calculated event rates
$= \{,\} = \{,\}$	all parameters
	free parameters
	constrained parameters
	parameters of interest
	nuisance parameters
$\kappa()$	multiplicative rate modifier
$\Delta()$	additive rate modifier
$c_(a\|)$	constraint term for constrained parameter
σ	relative uncertainty in the constrained parameter

CHAPTER

TWO

DECLARATIVE FORMATS

While flexible enough to describe a wide range of LHC measurements, the design of the specification is sufficiently simple to admit a *declarative format* that fully encodes the statistical model of the analysis. This format defines the channels, all associated samples, their parameterised rate modifiers and implied constraint terms as well as the measurements. Additionally, the format represents the mathematical model, leaving the implementation of the likelihood minimisation to be analysis-dependent and/or language-dependent. Originally XML was chosen as a specification language to define the structure of the model while introducing a dependence on to encode the nominal rates and required input data of the constraint terms [2]. Using this specification, a model can be constructed and evaluated within the framework.

This package introduces an updated form of the specification based on the ubiquitous plain-text JSON format and its schema-language *JSON Schema*. Described in more detail in *Likelihood Specification*, this schema fully specifies both structure and necessary constrained data in a single document and thus is implementation independent.

CHAPTER

THREE

ADDITIONAL MATERIAL

3.1 Footnotes

3.2 Bibliography

CHAPTER

FOUR

LIKELIHOOD SPECIFICATION

The structure of the JSON specification of models follows closely the original XML-based specification [2].

4.1 Workspace

```
{
    "$schema": "http://json-schema.org/draft-06/schema#",
    "$id": "https://diana-hep.org/pyhf/schemas/1.0.0/workspace.json",
    "type": "object",
    "properties": {
        "channels": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪channel"} },
        "measurements": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪measurement"} },
        "observations": { "type": "array", "items": {"$ref": "defs.json#/definitions/
↪observation" } },
        "version": { "const": "1.0.0" }
    },
    "additionalProperties": false,
    "required": ["channels", "measurements", "observations", "version"]
}
```

The overall document in the above code snippet describes a *workspace*, which includes

- **measurements**: The channels in the model, which include a description of the samples within each channel and their possible parametrised modifiers.
- **observations**: The observed data, with which a likelihood can be constructed from the
- **model**: A set of measurements, which define among others the parameters of interest for a given statistical analysis objective.

A workspace consists of the channels, one set of observed data, but can include multiple measurements. If provided a JSON file, one can quickly check that it conforms to the provided workspace specification as follows:

```
import json, requests, jsonschema
workspace = json.load(open('/path/to/analysis_workspace.json'))
# if no exception is raised, it found and parsed the schema
schema = requests.get('https://diana-hep.org/pyhf/schemas/1.0.0/workspace.json').
↪json()
# If no exception is raised by validate(), the instance is valid.
jsonschema.validate(instance=workspace, schema=schema)
```

4.2 Channel

A channel is defined by a channel name and a list of samples [1].

```
{
    "channel": {
        "type": "object",
        "properties": {
            "name": { "type": "string" },
            "samples": { "type": "array", "items": {"$ref": "#/definitions/sample"},
↪"minItems": 1 }
        },
        "required": ["name", "samples"],
        "additionalProperties": false
    },
}
```

The Channel specification consists of a list of channel descriptions. Each channel, an analysis region encompassing one or more measurement bins, consists of a `name` field and a `samples` field (see *Channel*), which holds a list of sample definitions (see *Sample*). Each sample definition in turn has a `name` field, a `data` field for the nominal event rates for all bins in the channel, and a `modifiers` field of the list of modifiers for the sample.

4.3 Sample

A sample is defined by a sample name, the sample event rate, and a list of modifiers [1].

```
{
    "sample": {
        "type": "object",
        "properties": {
            "name": { "type": "string" },
            "data": { "type": "array", "items": {"type": "number"}, "minItems": 1 },
            "modifiers": {
                "type": "array",
                "items": {
                    "anyOf": [
                        { "$ref": "#/definitions/modifier/histosys" },
                        { "$ref": "#/definitions/modifier/lumi" },
                        { "$ref": "#/definitions/modifier/normfactor" },
                        { "$ref": "#/definitions/modifier/normsys" },
                        { "$ref": "#/definitions/modifier/shapefactor" },
                        { "$ref": "#/definitions/modifier/shapesys" },
                        { "$ref": "#/definitions/modifier/staterror" }
                    ]
                }
            }
        },
        "required": ["name", "data", "modifiers"],
        "additionalProperties": false
    },
}
```

4.4 Modifiers

The modifiers that are applicable for a given sample are encoded as a list of JSON objects with three fields. A name field, a type field denoting the class of the modifier, and a data field which provides the necessary input data as denoted in *Modifiers and Constraints*.

Based on the declared modifiers, the set of parameters and their constraint terms are derived implicitly as each type of modifier unambiguously defines the constraint terms it requires. Correlated shape modifiers and normalisation uncertainties have compatible constraint terms and thus modifiers can be declared that *share* parameters by re-using a name[1] for multiple modifiers. That is, a variation of a single parameter causes a shift within sample rates due to both shape and normalisation variations.

We review the structure of each modifier type below.

4.4.1 Uncorrelated Shape (shapesys)

To construct the constraint term, the relative uncertainties σ_b are necessary for each bin. Therefore, we record the absolute uncertainty as an array of floats, which combined with the nominal sample data yield the desired σ_b. An example is shown below:

```
{ "name": "mod_name", "type": "shapesys", "data": [1.0, 1.5, 2.0] }
```

An example of an uncorrelated shape modifier with three absolute uncertainty terms for a 3-bin channel.

4.4.2 Correlated Shape (histosys)

This modifier represents the same source of uncertainty which has a different effect on the various sample shapes, hence a correlated shape. To implement an interpolation between sample distribution shapes, the distributions with a "downward variation" ("lo") associated with $\alpha = -1$ and an "upward variation" ("hi") associated with $\alpha = +1$ are provided as arrays of floats. An example is shown below:

```
{ "name": "mod_name", "type": "histosys", "data": {"hi_data": [20,15], "lo_data": [10,
↪ 10]} }
```

An example of a correlated shape modifier with absolute shape variations for a 2-bin channel.

4.4.3 Normalisation Uncertainty (normsys)

The normalisation uncertainty modifies the sample rate by a overall factor $\kappa(\alpha)$ constructed as the interpolation between downward ("lo") and upward ("hi") as well as the nominal setting, i.e. $\kappa(-1) = \kappa_{\alpha=-1}$, $\kappa(0) = 1$ and $\kappa(+1) = \kappa_{\alpha=+1}$. In the modifier definition we record $\kappa_{\alpha=+1}$ and $\kappa_{\alpha=-1}$ as floats. An example is shown below:

```
{ "name": "mod_name", "type": "normsys", "data": {"hi": 1.1, "lo": 0.9} }
```

An example of a normalisation uncertainty modifier with scale factors recorded for the up/down variations of an n-bin channel.

[1] The name of a modifier specifies the parameter set it is controlled by. Modifiers with the same name share parameter sets.

4.4.4 MC Statistical Uncertainty (staterror)

As the sample counts are often derived from Monte Carlo (MC) datasets, they necessarily carry an uncertainty due to the finite sample size of the datasets. As explained in detail in [2], adding uncertainties for each sample would yield a very large number of nuisance parameters with limited utility. Therefore a set of bin-wise scale factors γ_b is introduced to model the overall uncertainty in the bin due to MC statistics. The constrained term is constructed as a set of Gaussian constraints with a central value equal to unity for each bin in the channel. The scales σ_b of the constraint are computed from the individual uncertainties of samples defined within the channel relative to the total event rate of all samples: $\delta_{csb} = \sigma_{csb} / \sum_s \nu^0_{scb}$. As not all samples are within a channel are estimated from MC simulations, only the samples with a declared statistical uncertainty modifier enter the sum. An example is shown below:

```
{ "name": "mod_name", "type": "staterror", "data": [0.1] }
```

An example of a statistical uncertainty modifier.

4.4.5 Luminosity (lumi)

Sample rates derived from theory calculations, as opposed to data-driven estimates, are scaled to the integrated luminosity corresponding to the observed data. As the luminosity measurement is itself subject to an uncertainty, it must be reflected in the rate estimates of such samples. As this modifier is of global nature, no additional per-sample information is required and thus the data field is nulled. This uncertainty is relevant, in particular, when the parameter of interest is a signal cross-section. The luminosity uncertainty σ_λ is provided as part of the parameter configuration included in the measurement specification discussed in *Measurements*. An example is shown below:

```
{ "name": "mod_name", "type": "lumi", "data": null }
```

An example of a luminosity modifier.

4.4.6 Unconstrained Normalisation (normfactor)

The unconstrained normalisation modifier scales the event rates of a sample by a free parameter μ. Common use cases are the signal rate of a possible BSM signal or simultaneous in-situ measurements of background samples. Such parameters are frequently the parameters of interest of a given measurement. No additional per-sample data is required. An example is shown below:

```
{ "name": "mod_name", "type": "normfactor", "data": null }
```

An example of a normalisation modifier.

4.4.7 Data-driven Shape (shapefactor)

In order to support data-driven estimation of sample rates (e.g. for multijet backgrounds), the data-driven shape modifier adds free, bin-wise multiplicative parameters. Similarly to the normalisation factors, no additional data is required as no constraint is defined. An example is shown below:

```
{ "name": "mod_name", "type": "shapefactor", "data": null }
```

An example of an uncorrelated shape modifier.

4.5 Data

The data provided by the analysis are the observed data for each channel (or region). This data is provided as a mapping from channel name to an array of floats, which provide the observed rates in each bin of the channel. The auxiliary data is not included as it is an input to the likelihood that does not need to be archived and can be determined automatically from the specification. An example is shown below:

```
{ "chan_name_one": [10, 20], "chan_name_two": [4, 0]}
```

An example of channel data.

4.6 Measurements

Given the data and the model definitions, a measurement can be defined. In the current schema, the measurements defines the name of the parameter of interest as well as parameter set configurations.[2] Here, the remaining information not covered through the channel definition is provided, e.g. for the luminosity parameter. For all modifiers, the default settings can be overridden where possible:

- **inits**: Initial value of the parameter.
- **bounds**: Interval bounds of the parameter.
- **auxdata**: Auxiliary data for the associated constraint term.
- **sigmas**: Associated uncertainty of the parameter.

An example is shown below:

```
{
    "name": "MyMeasurement",
    "config": {
        "poi": "SignalCrossSection", "parameters": [
            { "name":"lumi", "auxdata":[1.0],"sigmas":[0.017], "bounds":[[0.915,1.
↪085]],"inits":[1.0] },
            { "name":"mu_ttbar", "bounds":[[0, 5]] },
            { "name":"rw_1CR", "fixed":true }
        ]
    }
}
```

An example of a measurement. This measurement, which scans over the parameter of interest `SigXSec`, is setting configurations for the luminosity modifier, changing the default bounds for the normfactor modifier named `alpha_ttbar`, and specifying that the modifier `rw_1CR` is held constant (`fixed`).

[2] In this context a parameter set corresponds to a named lower-dimensional subspace of the full parameters . In many cases these are one-dimensional subspaces, e.g. a specific interpolation parameter α or the luminosity parameter λ. For multi-bin channels, however, e.g. all bin-wise nuisance parameters of the uncorrelated shape modifiers are grouped under a single name. Therefore in general a parameter set definition provides arrays of initial values, bounds, etc.

4.7 Observations

This is what we evaluate the hypothesis testing against, to determine the compatibility of signal+background hypothesis to the background-only hypothesis. This is specified as a list of objects, with each object structured as

- **name**: the channel for which the observations are recorded
- **data**: the bin-by-bin observations for the named channel

An example is shown below:

```
{
    "name": "channel1",
    "data": [110.0, 120.0]
}
```

An example of an observation. This observation recorded for a 2-bin channel `channel1`, has values `110.0` and `120.0`.

4.8 Toy Example

```
{
    "channels": [
        { "name": "singlechannel",
          "samples": [
            { "name": "signal",
              "data": [5.0, 10.0],
              "modifiers": [ { "name": "mu", "type": "normfactor", "data": null} ]
            },
            { "name": "background",
              "data": [50.0, 60.0],
              "modifiers": [ {"name": "uncorr_bkguncrt", "type": "shapesys", "data":␣
↪[5.0, 12.0]} ]
            }
          ]
        }
    ],
    "observations": [
        { "name": "singlechannel", "data": [50.0, 60.0] }
    ],
    "measurements": [
        { "name": "Measurement", "config": {"poi": "mu", "parameters": []} }
    ],
    "version": "1.0.0"
}
```

In the above example, we demonstrate a simple measurement of a single two-bin channel with two samples: a signal sample and a background sample. The signal sample has an unconstrained normalisation factor μ, while the background sample carries an uncorrelated shape systematic controlled by parameters γ_1 and γ_2. The background uncertainty for the bins is 10% and 20% respectively.

4.9 Additional Material

4.9.1 Footnotes

4.9.2 Bibliography

CHAPTER

FIVE

EXAMPLES

Try out in Binder!

Notebooks:

5.1 Hello World, `pyhf` style

Two bin counting experiment with a background uncertainty

[1]:
```
import pyhf
```

Returning the observed and expected CL_s

[2]:
```
pdf = pyhf.simplemodels.hepdata_like(signal_data=[12.0, 11.0], bkg_data=[50.0, 52.0],␣
↪bkg_uncerts=[3.0, 7.0])
CLs_obs, CLs_exp = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected=True)
print('Observed: {}, Expected: {}'.format(CLs_obs, CLs_exp))
```

```
Observed: [0.05290116], Expected: [0.06445521]
```

Returning the observed CL_s, CL_{s+b}, and CL_b

[3]:
```
CLs_obs, p_values = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_tail_probs=True)
print('Observed CL_s: {}, CL_sb: {}, CL_b: {}'.format(CLs_obs, p_values[0], p_
↪values[1]))
```

```
Observed CL_s: [0.05290116], CL_sb: [0.0236], CL_b: [0.44611493]
```

A reminder that

$$\mathrm{CL}_s = \frac{\mathrm{CL}_{s+b}}{\mathrm{CL}_b} = \frac{p_{s+b}}{1-p_b}$$

[4]:
```
assert CLs_obs == p_values[0]/p_values[1]
```

Returning the expected CL_s band values

[5]:
```
import numpy as np
```

[6]:
```
CLs_obs, CLs_exp_band = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected_set=True)
print('Observed CL_s: {}\n'.format(CLs_obs))
for p_value, n_sigma in enumerate(np.arange(-2,3)):
    print('Expected CL_s{}: {}'.format('      ' if n_sigma==0 else '({} )'.format(n_
↪sigma),CLs_exp_band[p_value]))
```

```
Observed CL_s: [0.05290116]

Expected CL_s(-2 ): [0.00260641]
Expected CL_s(-1 ): [0.01382066]
Expected CL_s      : [0.06445521]
Expected CL_s(1 ): [0.23526104]
Expected CL_s(2 ): [0.57304182]
```

Returning the test statistics for the observed and Asimov data

[7]:
```
CLs_obs, test_statistics = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata,␣
↪pdf, return_test_statistics=True)
print('q_mu: {}, Asimov q_mu: {}'.format(test_statistics[0], test_statistics[1]))
```

```
q_mu: [3.93824492], Asimov q_mu: [3.41886758]
```

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
import os
import pyhf
import pyhf.readxml
from ipywidgets import interact, fixed
```

5.2 Binned HEP Statistical Analysis in Python

5.2.1 HistFactory

HistFactory is a popular framework to analyze binned event data and commonly used in High Energy Physics. At its core it is a template for building a statistical model from individual binned distribution ('Histograms') and variations on them ('Systematics') that represent auxiliary measurements (for example an energy scale of the detector which affects the shape of a distribution)

5.2.2 pyhf

`pyhf` is a work-in-progress standalone implementation of the HistFactory p.d.f. template and an implementation of the test statistics and asymptotic formulae described in the paper by Cowan, Cranmer, Gross, Vitells: *Asymptotic formulae for likelihood-based tests of new physics* [arxiv:1007.1727].

Models can be defined using JSON specification, but existing models based on the XML + ROOT file scheme are readable as well.

5.2.3 The Demo

The input data for the statistical analysis was built generated using the containerized workflow engine yadage (see demo from KubeCon 2018 [youtube]). Similarly to Binder this utilizes modern container technology for reproducible science. Below you see the execution graph leading up to the model input data at the bottom.

[3]:
```
import base64
from IPython.core.display import display, HTML
anim = base64.b64encode(open('workflow.gif','rb').read()).decode('ascii')
HTML('<img src="{}">'.format(anim))
```

[3]:
```
<IPython.core.display.HTML object>
```

5.2.4 Read in the Model from XML and ROOT

The ROOT files are read using scikit-hep's uproot module.

[4]:
```
parsed = pyhf.readxml.parse('meas.xml',os.getcwd())
workspace = pyhf.Workspace(parsed)
obs_data = workspace.observations['channel1']
```

From the parsed data, we construct a probability density function (p.d.f). As the model includes systematics a number of implied "auxiliary measurements" must be added to the observed data distribution.

[5]:
```
pdf = pyhf.Model({'channels': parsed['channels'], 'parameters': parsed['measurements
↪'][0]['config']['parameters']}, poiname = 'SigXsecOverSM')
data = obs_data + pdf.config.auxdata
```

The p.d.f is build from one data-drived "qcd" (or multijet) estimate and two Monte Carlo-based background samples and is parametrized by five parameters: One parameter of interest `SigXsecOverSM` and four *nuisance parameters* that affect the shape of the two Monte Carlo background estimates (both weight-only and shape systematics)

[6]:
```
par_name_dict = {k: v['slice'].start for k,v in pdf.config.par_map.items()}
print('Samples:\n {}'.format(pdf.config.samples))
print('Parameters:\n {}'.format(par_name_dict))
```

```
Samples:
 ['mc1', 'mc2', 'qcd', 'signal']
Parameters:
 {'lumi': 0, 'SigXsecOverSM': 1, 'mc1_weight_var1': 2, 'mc1_shape_conv': 3, 'mc2_
↪weight_var1': 4, 'mc2_shape_conv': 5}
```

[7]:
```
all_par_settings = {n[0]: tuple(m) for n,m in zip(sorted(reversed(list(par_name_dict.
↪items())), key=lambda x:x[1]), pdf.config.suggested_bounds())}
default_par_settings = {n[0]: sum(tuple(m))/2.0 for n,m in all_par_settings.items()}

def get_mc_counts(pars):
    deltas, factors = pdf._modifications(pars)
    allsum = pyhf.tensorlib.concatenate(deltas + [pyhf.tensorlib.astensor(pdf.
↪thenom)])
    nom_plus_delta = pyhf.tensorlib.sum(allsum,axis=0)
    nom_plus_delta = pyhf.tensorlib.reshape(nom_plus_delta,(1,)+pyhf.tensorlib.
↪shape(nom_plus_delta))
    allfac = pyhf.tensorlib.concatenate(factors + [nom_plus_delta])
    return pyhf.tensorlib.product(allfac,axis=0)
```

(continues on next page)

(continued from previous page)

```
animate_plot_pieces = None
def init_plot(fig, ax, par_settings):
    global animate_plot_pieces

    nbins = sum(list(pdf.config.channel_nbins.values()))
    x = np.arange(nbins)
    data = np.zeros(nbins)
    items = []
    for i in [3, 2, 1, 0]:
        items.append(ax.bar(x, data, 1, alpha=1.0))
    animate_plot_pieces = (items, ax.scatter(x, obs_data, c='k', alpha=1., zorder=99))

def animate(ax=None, fig=None, **par_settings):
    global animate_plot_pieces
    items, obs = animate_plot_pieces
    pars = pyhf.tensorlib.astensor(pdf.config.suggested_init())
    for k,v in par_settings.items():
        pars[par_name_dict[k]] = v

    mc_counts = get_mc_counts(pars)
    rectangle_collection = zip(*map(lambda x: x.patches, items))

    for rectangles,binvalues in zip(rectangle_collection, mc_counts[:,0].T):
        offset = 0
        for sample_index in [3, 2, 1, 0]:
            rect = rectangles[sample_index]
            binvalue = binvalues[sample_index]
            rect.set_y(offset)
            rect.set_height(binvalue)
            offset += rect.get_height()

    fig.canvas.draw()

def plot(ax=None, order=[3, 2, 1, 0], **par_settings):
    pars = pyhf.tensorlib.astensor(pdf.config.suggested_init())
    for k,v in par_settings.items():
        pars[par_name_dict[k]] = v

    mc_counts = get_mc_counts(pars)
    bottom = None
    # nb: bar_data[0] because evaluating only one parset
    for i,sample_index in enumerate(order):
        data = mc_counts[sample_index][0]
        x = np.arange(len(data))
        ax.bar(x, data, 1, bottom = bottom, alpha = 1.0)
        bottom = data if i==0 else bottom + data
    ax.scatter(x, obs_data, c = 'k', alpha = 1., zorder=99)
```

5.2.5 Interactive Exploration of a HistFactory Model

One advantage of a pure-python implementation of Histfactory is the ability to explore the pdf interactively within the setting of a notebook. Try moving the sliders and oberserve the effect on the samples. For example changing the parameter of interest `SigXsecOverSM` (or μ) controls the overall normalization of the (BSM) signal sample (μ=0 for background-only and μ=1 for the nominal signal-plus-background hypothesis)

[8]:
```
%matplotlib notebook
fig, ax = plt.subplots(1, 1)
fig.set_size_inches(10, 5)
ax.set_ylim(0, 1.5 * np.max(obs_data))

init_plot(fig, ax, default_par_settings)
interact(animate, fig=fixed(fig), ax=fixed(ax), **all_par_settings);
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
interactive(children=(FloatSlider(value=1.0, description='lumi', max=1.5, min=0.5),␣
↪IntSlider(value=5, descrip...
```

[9]:
```
nominal = pdf.config.suggested_init()
background_only = pdf.config.suggested_init()
background_only[pdf.config.poi_index] = 0.0
best_fit = pyhf.optimizer.unconstrained_bestfit(
    pyhf.utils.loglambdav, data, pdf, pdf.config.suggested_init(), pdf.config.
↪suggested_bounds())
```

```
/Users/kratsg/pyhf/pyhf/tensor/numpy_backend.py:184: RuntimeWarning: invalid value␣
↪encountered in log
  return n * np.log(lam) - lam - gammaln(n + 1.0)
```

5.2.6 Fitting

We can now fit the statistical model to the observed data. The best fit of the signal strength is close to the background-only hypothesis.

[10]:
```
f,(ax1,ax2,ax3) = plt.subplots(1,3, sharey=True, sharex=True)
f.set_size_inches(18,4)
ax1.set_ylim(0,1.5*np.max(obs_data))
ax1.set_title(u'nominal signal + background μ = 1')
plot(ax = ax1, **{k: nominal[v] for k,v in par_name_dict.items()})

ax2.set_title(u'nominal background-only μ = 0')
plot(ax = ax2, **{k: background_only[v] for k,v in par_name_dict.items()})

ax3.set_title(u'best fit μ = {:.3g}'.format(best_fit[pdf.config.poi_index]))
plot(ax = ax3, **{k: best_fit[v] for k,v in par_name_dict.items()})
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

5.2.7 Interval Estimation (Computing Upper Limits on μ)

A common task in the statistical evaluation of High Energy Physics data analyses is the estimation of confidence intervals of parameters of interest. The general strategy is to perform a series of hypothesis tests and then *invert* the tests in order to obtain an interval with the correct coverage properties.

A common figure of merit is a modified p-value, CLs. Here we compute an upper limit based on a series of CLs tests.

[11]:
```
def plot_results(ax, test_mus, cls_obs, cls_exp, test_size=0.05):
    ax.plot(mu_tests, cls_obs, c = 'k')
    for i,c in zip(range(5),['k','k','k','k','k']):
        ax.plot(mu_tests, cls_exp[i], c = c, linestyle = 'dotted' if i!=2 else 'dashed
↪')
    ax.fill_between(test_mus,cls_exp[0],cls_exp[-1], facecolor = 'y')
    ax.fill_between(test_mus,cls_exp[1],cls_exp[-2], facecolor = 'g')
    ax.plot(test_mus,[test_size]*len(test_mus), c = 'r')
    ax.set_ylim(0,1)

def invert_interval(test_mus, cls_obs, cls_exp, test_size = 0.05):
    point05cross = {'exp':[],'obs':None}
    for cls_exp_sigma in cls_exp:
        y_vals = cls_exp_sigma
        point05cross['exp'].append(np.interp(test_size, list(reversed(y_vals)),␣
↪list(reversed(test_mus))))
    yvals = cls_obs
    point05cross['obs'] = np.interp(test_size, list(reversed(y_vals)),␣
↪list(reversed(test_mus)))
    return point05cross
```

[12]:
```
mu_tests = np.linspace(0, 1, 16)
hypo_tests = [pyhf.utils.hypotest(mu, data, pdf, pdf.config.suggested_init(), pdf.
↪config.suggested_bounds(),
                                  return_expected_set=True, return_test_
↪statistics=True)
              for mu in mu_tests]

test_stats = np.array([test[-1][0] for test in hypo_tests]).flatten()
cls_obs = np.array([test[0] for test in hypo_tests]).flatten()
cls_exp = [np.array([test[1][i] for test in hypo_tests]).flatten() for i in range(5)]

fig, (ax1,ax2) = plt.subplots(1, 2)
fig.set_size_inches(15, 5)

ax1.set_title(u'Hypothesis Tests')
ax1.set_ylabel(u'CLs')
ax1.set_xlabel(u'μ')
plot_results(ax1, mu_tests, cls_obs, cls_exp)


ax2.set_title(u'Test Statistic')
ax2.set_xlabel(u'μ')
ax2.plot(mu_tests,test_stats);
```

```
/Users/kratsg/pyhf/pyhf/tensor/numpy_backend.py:184: RuntimeWarning: invalid value␣
↪encountered in log
  return n * np.log(lam) - lam - gammaln(n + 1.0)
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

[13]:
```
results = invert_interval(mu_tests, cls_obs, cls_exp)

print('Observed Limit: {:.2f}'.format(results['obs']))
print('-----')
for i, n_sigma in enumerate(np.arange(-2,3)):
    print('Expected Limit{}: {:.3f}'.format('' if n_sigma==0 else '({} )'.format(n_
↪sigma),results['exp'][i]))
```

```
Observed Limit: 0.96
-----
Expected Limit(-2 ): 0.266
Expected Limit(-1 ): 0.363
Expected Limit: 0.505
Expected Limit(1 ): 0.707
Expected Limit(2 ): 0.956
```

5.3 XML Import/Export

[1]:
```
# NB: pip install pyhf[xmlio]
import pyhf
```

[2]:
```
!ls -lavh ../../../validation/xmlimport_input
```

```
total 1752
drwxr-xr-x   7 kratsg  staff   238B Oct 16 22:20 .
drwxr-xr-x  21 kratsg  staff   714B Apr  4 14:26 ..
drwxr-xr-x   6 kratsg  staff   204B Feb 27 17:13 config
drwxr-xr-x   7 kratsg  staff   238B Feb 27 23:41 data
-rw-r--r--   1 kratsg  staff   850K Oct 16 22:20 log
drwxr-xr-x  17 kratsg  staff   578B Nov 15 12:24 results
-rw-r--r--   1 kratsg  staff    21K Oct 16 22:20 scan.pdf
```

5.3.1 Importing

In order to convert HistFactory XML+ROOT to the pyhf JSON spec for likelihoods, you need to point the command-line interface `pyhf xml2json` at the top-level XML file. Additionally, as the HistFactory XML specification often uses relative paths, you might need to specify the base directory `--basedir` from which all other files are located, as specified in the top-level XML. The command will be of the format

```
pyhf xml2json {top-level XML} --basedir {base directory}
```

This will print the JSON representation of the XML+ROOT specified. If you wish to store this as a JSON file, you simply need to redirect it

```
pyhf xml2json {top-level XML} --basedir {base directory} > spec.json
```

[3]:
```
!pyhf xml2json --hide-progress ../../../validation/xmlimport_input/config/example.xml␣
↪--basedir ../../../validation/xmlimport_input | tee xml_importexport.json
```

```
{
    "channels": [
        {
            "name": "channel1",
            "samples": [
                {
                    "data": [
                        20.0,
                        10.0
                    ],
                    "modifiers": [
                        {
                            "data": {
                                "hi": 1.05,
                                "lo": 0.95
                            },
                            "name": "syst1",
                            "type": "normsys"
                        },
                        {
                            "data": null,
                            "name": "SigXsecOverSM",
                            "type": "normfactor"
                        }
                    ],
                    "name": "signal"
                },
                {
                    "data": [
                        100.0,
                        0.0
                    ],

                        {
                            "data": null,
                            "name": "lumi",
                            "type": "lumi"
                        },
                        {
                            "data": [
                                5.000000074505806,
                                0.0
                            ],
                            "name": "staterror_channel1",
                            "type": "staterror"
                        },
                        {
                            "data": {
                                "hi": 1.05,
                                "lo": 0.95
                            },
                            "name": "syst2",
                            "type": "normsys"
                        }
                    ],
                    "name": "background1"
                },
```


(continues on next page)

(continued from previous page)


```
                {
                    "data": [
                        0.0,
                        100.0
                    ],
                    "modifiers": [
                        {
                            "data": null,
                            "name": "lumi",
                            "type": "lumi"
                        },
                        {
                            "data": [
                                0.0,
                                10.0
                            ],
                            "name": "staterror_channel1",
                            "type": "staterror"
                        },
                        {
                            "data": {
                                "hi": 1.05,
                                "lo": 0.95
                            },
                            "name": "syst3",
                            "type": "normsys"
                        }
                    ],
                    "name": "background2"
                }
            ]
        }
    ],
    "data": {
        "channel1": [
            122.0,
            112.0
        ]
    },
    "toplvl": {
        "measurements": [
            {
                "config": {
                    "parameters": [
                        {
                            "auxdata": [
                                1.0
                            ],
                            "bounds": [
                                [
                                    0.5,
                                    1.5
                                ]
                            ],
                            "fixed": true,
                            "inits": [
                                1.0
```


(continues on next page)

(continued from previous page)


```
                    ],
                    "name": "lumi",
                    "sigmas": [
                        0.1
                    ]
                },
                {
                    "fixed": true,
                    "name": "alpha_syst1"
                }
            ],
            "poi": "SigXsecOverSM"
        },
        "name": "GaussExample"
    },
    {
        "config": {
            "parameters": [
                {
                    "auxdata": [
                        1.0
                    ],
                    "bounds": [
                        [
                            0.5,
                            1.5
                        ]
                    ],
                    "fixed": true,
                    "inits": [
                        1.0
                    ],
                    "name": "lumi",
                    "sigmas": [
                        0.1
                    ]
                },
                {
                    "fixed": true,
                    "name": "alpha_syst1"
                }
            ],
            "poi": "SigXsecOverSM"
        },
        "name": "GammaExample"
    },
    {
        "config": {
            "parameters": [
                {
                    "auxdata": [
                        1.0
                    ],
                    "bounds": [
                        [
                            0.5,
                            1.5
```


(continues on next page)

(continued from previous page)


```
                                ]
                            ],
                            "fixed": true,
                            "inits": [
                                1.0
                            ],
                            "name": "lumi",
                            "sigmas": [
                                0.1
                            ]
                        },
                        {
                            "fixed": true,
                            "name": "alpha_syst1"
                        }
                    ],
                    "poi": "SigXsecOverSM"
                },
                "name": "LogNormExample"
            },
            {
                "config": {
                    "parameters": [
                        {
                            "auxdata": [
                                1.0
                            ],
                            "bounds": [
                                [
                                    0.5,
                                    1.5
                                ]
                            ],
                            "fixed": true,
                            "inits": [
                                1.0
                            ],
                            "name": "lumi",
                            "sigmas": [
                                0.1
                            ]
                        },
                        {
                            "fixed": true,
                            "name": "alpha_syst1"
                        }
                    ],
                    "poi": "SigXsecOverSM"
                },
                "name": "ConstExample"
            }
        ],
        "resultprefix": "./results/example"
    }
}
```

5.3.2 Exporting

In order to convert the pyhf JSON to the HistFactory XML+ROOT spec for likelihoods, you need to point the command-line interface `pyhf json2xml` at the JSON file you want to convert. As everything is specified in a single file, there is no need to deal with base directories or looking up additional files. This will produce output XML+ROOT in the `--output-dir=./` directory (your current working directory), storing XML configs under `--specroot=config` and the data file under `--dataroot=data`. The XML configs are prefixed with `--resultprefix=FitConfig` by default, so that the top-level XML file will be located at `{output dir}/{prefix}.xml`. The command will be of the format

```
pyhf json2xml {JSON spec}
```

Note that the output directory must already exist.

[4]:
```
!mkdir -p output
!pyhf json2xml xml_importexport.json --output-dir output
!ls -lavh output/*
```

```
/Users/kratsg/pyhf/pyhf/writexml.py:120: RuntimeWarning: invalid value encountered in␣
↪true_divide
  attrs['HistoName'], np.divide(modifierspec['data'], sampledata).tolist()
-rw-r--r--  1 kratsg  staff   822B Apr  9 09:36 output/FitConfig.xml

output/config:
total 8
drwxr-xr-x  3 kratsg  staff   102B Apr  9 09:36 .
drwxr-xr-x  5 kratsg  staff   170B Apr  9 09:36 ..
-rw-r--r--  1 kratsg  staff   1.0K Apr  9 09:36 FitConfig_channel1.xml

output/data:
total 96
drwxr-xr-x  3 kratsg  staff   102B Apr  9 09:36 .
drwxr-xr-x  5 kratsg  staff   170B Apr  9 09:36 ..
-rw-r--r--  1 kratsg  staff    46K Apr  9 09:36 data.root
```

[5]:
```
!rm xml_importexport.json
!rm -rf output/
```

5.4 ShapeFactor

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
import logging
import json

import pyhf
from pyhf import Model

logging.basicConfig(level = logging.INFO)
```

[3]:
```
def prep_data(sourcedata):
    spec =  {
```

(continues on next page)

(continued from previous page)

```
        'channels': [
            {
                'name': 'signal',
                'samples': [
                    {
                        'name': 'signal',
                        'data': sourcedata['signal']['bindata']['sig'],
                        'modifiers': [
                            {
                                'name': 'mu',
                                'type': 'normfactor',
                                'data': None
                            }
                        ]
                    },
                    {
                        'name': 'bkg1',
                        'data': sourcedata['signal']['bindata']['bkg1'],
                        'modifiers': [
                            {
                                'name': 'coupled_shapefactor',
                                'type': 'shapefactor',
                                'data': None
                            }
                        ]
                    }
                ]
            },
            {
                'name': 'control',
                'samples': [
                    {
                        'name': 'background',
                        'data': sourcedata['control']['bindata']['bkg1'],
                        'modifiers': [
                            {
                                'name': 'coupled_shapefactor',
                                'type': 'shapefactor',
                                'data': None
                            }
                        ]
                    }
                ]
            }
        ]
    }
    pdf  = Model(spec)
    data = []
    for c in pdf.spec['channels']:
        data += sourcedata[c['name']]['bindata']['data']
    data = data + pdf.config.auxdata
    return data, pdf
```

[4]:
```
source = {
  "channels": {
    "signal": {
```

(continues on next page)

(continued from previous page)

```
      "binning": [2,-0.5,1.5],
      "bindata": {
        "data":     [220.0, 230.0],
        "bkg1":     [100.0, 70.0],
        "sig":      [ 20.0, 20.0]
      }
    },
    "control": {
      "binning": [2,-0.5,1.5],
      "bindata": {
        "data":    [200.0, 300.0],
        "bkg1":    [100.0, 100.0]
      }
    }
  }
}

data, pdf = prep_data(source['channels'])
print('data: {}'.format(data))

init_pars = pdf.config.suggested_init()
print('expected data: {}'.format(pdf.expected_data(init_pars)))

par_bounds = pdf.config.suggested_bounds()
```

```
INFO:pyhf.pdf:Validating spec against schema: /home/jovyan/pyhf/pyhf/data/spec.json
INFO:pyhf.pdf:adding modifier mu (1 new nuisance parameters)
INFO:pyhf.pdf:adding modifier coupled_shapefactor (2 new nuisance parameters)
```

```
data: [220.0, 230.0, 200.0, 300.0]
expected data: [120.  90. 100. 100.]
```

[5]:
```
print('initialization parameters: {}'.format(pdf.config.suggested_init()))

unconpars = pyhf.optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, data, pdf,
                                                 pdf.config.suggested_init(), pdf.
↪config.suggested_bounds())
print('parameters post unconstrained fit: {}'.format(unconpars))
```

```
initialization parameters: [1.0, 1.0, 1.0]
parameters post unconstrained fit: [0.99981412 2.00002042 3.00006469]
```

```
/home/jovyan/pyhf/pyhf/tensor/numpy_backend.py:173: RuntimeWarning: divide by zero␣
↪encountered in log
  return n * np.log(lam) - lam - gammaln(n + 1.0)
```

[6]:
```
def plot_results(testmus, cls_obs, cls_exp, poi_tests, test_size = 0.05):
    plt.plot(poi_tests,cls_obs, c = 'k')
    for i,c in zip(range(5),['grey','grey','grey','grey','grey']):
        plt.plot(poi_tests, cls_exp[i], c = c)
    plt.plot(testmus,[test_size]*len(testmus), c = 'r')
    plt.ylim(0,1)

def invert_interval(testmus, cls_obs, cls_exp, test_size = 0.05):
    point05cross = {'exp':[],'obs':None}
    for cls_exp_sigma in cls_exp:
        yvals = cls_exp_sigma
        point05cross['exp'].append(np.interp(test_size,
```

(continues on next page)

(continued from previous page)

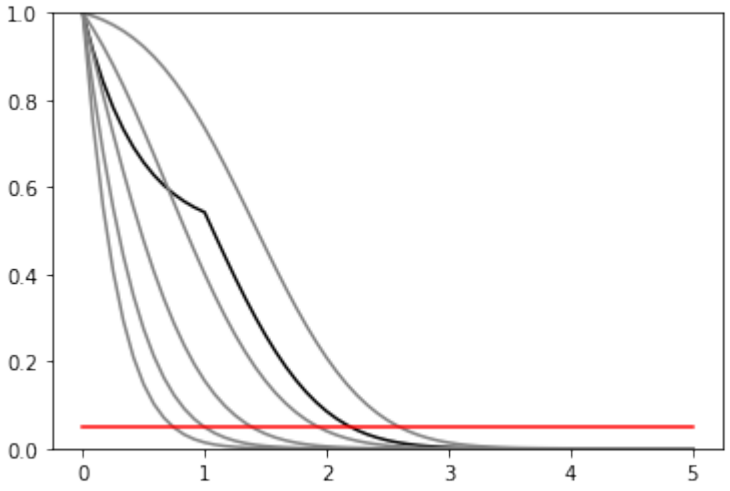
```
                                          list(reversed(yvals)),
                                          list(reversed(testmus))))

    yvals = cls_obs
    point05cross['obs'] = np.interp(test_size,
                                    list(reversed(yvals)),
                                    list(reversed(testmus)))
    return point05cross


poi_tests = np.linspace(0, 5, 61)
tests = [pyhf.utils.hypotest(poi_test, data, pdf, init_pars, par_bounds, return_
↪expected_set=True)
         for poi_test in poi_tests]
cls_obs = np.array([test[0] for test in tests]).flatten()
cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]

print('\n')
plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
invert_interval(poi_tests, cls_obs, cls_exp)
```

[6]:
```
{'exp': [0.741381412468345,
  0.9949353526744877,
  1.3845144105754894,
  1.9289946435921614,
  2.594077794516857],
 'obs': 2.1945970333027187}
```



5.5 Multi-bin Poisson

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
import logging
import json

import pyhf
from pyhf import Model, optimizer
from pyhf.simplemodels import hepdata_like

from scipy.interpolate import griddata
import scrapbook as sb
```

[3]:
```
def plot_results(testmus, cls_obs, cls_exp, poi_tests, test_size = 0.05):
    plt.plot(poi_tests,cls_obs, c = 'k')
    for i,c in zip(range(5),['grey','grey','grey','grey','grey']):
        plt.plot(poi_tests, cls_exp[i], c = c)
    plt.plot(testmus,[test_size]*len(testmus), c = 'r')
    plt.ylim(0,1)

def invert_interval(testmus, cls_obs, cls_exp, test_size = 0.05):
    point05cross = {'exp':[],'obs':None}
    for cls_exp_sigma in cls_exp:
        yvals = cls_exp_sigma
        point05cross['exp'].append(np.interp(test_size,
                                             list(reversed(yvals)),
                                             list(reversed(testmus))))

    yvals = cls_obs
    point05cross['obs'] = np.interp(test_size,
                                    list(reversed(yvals)),
                                    list(reversed(testmus)))
    return point05cross

def plot_histo(ax, binning, data):
    bin_width = (binning[2]-binning[1])/binning[0]
    bin_leftedges = np.linspace(binning[1],binning[2],binning[0]+1)[:-1]
    bin_centers = [le + bin_width/2. for le in bin_leftedges]
    ax.bar(bin_centers,data,1, alpha=0.5)


def plot_data(ax, binning, data):
    errors = [math.sqrt(d) for d in data]
    bin_width = (binning[2]-binning[1])/binning[0]
    bin_leftedges = np.linspace(binning[1],binning[2],binning[0]+1)[:-1]
    bin_centers = [le + bin_width/2. for le in bin_leftedges]
    ax.bar(bin_centers,data,0, yerr=errors, linewidth=0, error_kw = dict(ecolor='k',␣
↪elinewidth = 1))
    ax.scatter(bin_centers, data, c = 'k')
```

[4]:
```
validation_datadir = '../../validation/data'
```

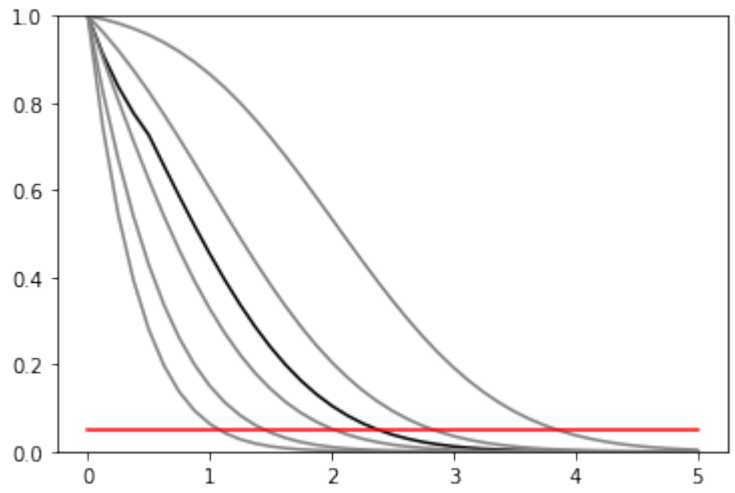
[5]:
```
source = json.load(open(validation_datadir + '/1bin_example1.json'))
pdf = hepdata_like(source['bindata']['sig'], source['bindata']['bkg'], source['bindata
↪']['bkgerr'])
data = source['bindata']['data'] + pdf.config.auxdata

init_pars  = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

poi_tests = np.linspace(0, 5, 41)
tests = [pyhf.utils.hypotest(poi_test, data, pdf, init_pars, par_bounds, return_
↪expected_set=True)
         for poi_test in poi_tests]
cls_obs = np.array([test[0] for test in tests]).flatten()
cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]

plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
invert_interval(poi_tests, cls_obs, cls_exp)
```

[5]:
```
{'exp': [1.0810606780537388,
  1.4517179965651292,
  2.0200754881420266,
  2.834863384648174,
  3.8487567494315487],
 'obs': 2.3800254370628036}
```



[6]:
```
source = {
  "binning": [2,-0.5,1.5],
  "bindata": {
    "data":    [120.0, 145.0],
    "bkg":     [100.0, 150.0],
    "bkgerr":  [15.0, 20.0],
    "sig":     [30.0, 45.0]
  }
}


my_observed_counts = source['bindata']['data']

pdf = hepdata_like(source['bindata']['sig'], source['bindata']['bkg'], source['bindata
↪']['bkgerr'])
```

(continues on next page)

(continued from previous page)

```
data = my_observed_counts + pdf.config.auxdata


binning = source['binning']

nompars = pdf.config.suggested_init()


bonlypars = [x for x in nompars]
bonlypars[pdf.config.poi_index] = 0.0
nom_bonly = pdf.expected_data(bonlypars, include_auxdata = False)

nom_sb = pdf.expected_data(nompars, include_auxdata = False)

init_pars = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

print(init_pars)

bestfit_pars = optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, data, pdf, init_
↪pars, par_bounds)
bestfit_cts  = pdf.expected_data(bestfit_pars, include_auxdata = False)
```

```
[1.0, 1.0, 1.0]
```

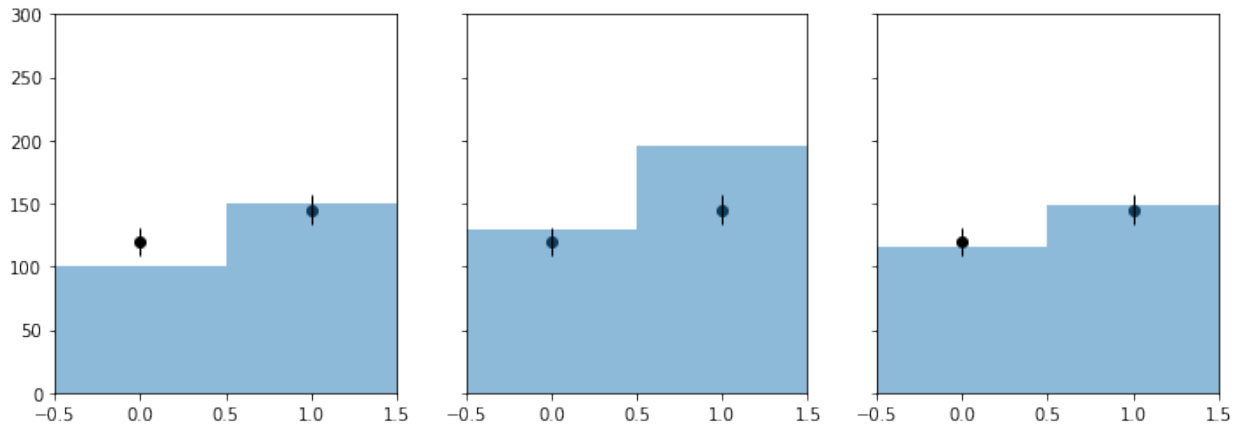
[7]:
```
f, axarr = plt.subplots(1,3,sharey=True)
f.set_size_inches(12,4)

plot_histo(axarr[0], binning, nom_bonly)
plot_data(axarr[0], binning, my_observed_counts)
axarr[0].set_xlim(binning[1:])

plot_histo(axarr[1], binning, nom_sb)
plot_data(axarr[1], binning, my_observed_counts)
axarr[1].set_xlim(binning[1:])

plot_histo(axarr[2], binning, bestfit_cts)
plot_data(axarr[2], binning, my_observed_counts)
axarr[2].set_xlim(binning[1:])

plt.ylim(0,300);
```



[8]:
```
##
##  DUMMY 2D thing
##

def signal(m1, m2):
    massscale = 150.
    minmass = 100.
    countscale = 2000

    effective_mass = np.sqrt(m1**2 + m2**2)
    return [countscale*np.exp(-(effective_mass-minmass)/massscale), 0]

def CLs(m1,m2):
    signal_counts = signal(m1, m2)
    pdf = hepdata_like(signal_counts, source['bindata']['bkg'], source['bindata'][
↪'bkgerr'])
    try:
        cls_obs, cls_exp_set = pyhf.utils.hypotest(1.0, data, pdf, init_pars, par_
↪bounds, return_expected_set=True)
        return cls_obs, cls_exp_set, True
    except AssertionError:
        print('fit failed for mass points ({}, {})'.format(m1, m2))
        return None, None, False
```

[9]:
```
nx, ny = 15, 15
grid = grid_x, grid_y = np.mgrid[100:1000:complex(0, nx), 100:1000:complex(0, ny)]
X = grid.T.reshape(nx * ny, 2)
results = [CLs(m1, m2) for m1, m2 in X]
```

[10]:
```
X = np.array([x for x,(_,_,success) in zip(X,results) if success])
yobs = np.array([obs for obs, exp, success in results if success]).flatten()
yexp = [np.array([exp[i] for obs, exp, success in results if success]).flatten() for␣
↪i in range(5)]
```

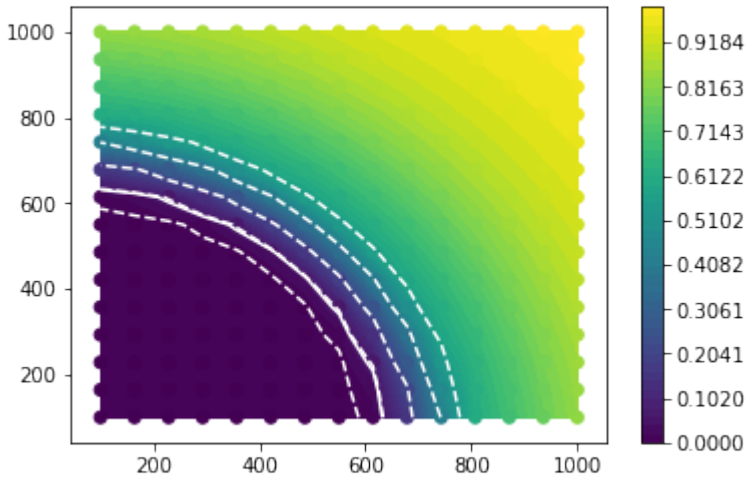
[11]:
```
int_obs = griddata(X, yobs, (grid_x, grid_y), method='linear')

int_exp = [griddata(X, yexp[i], (grid_x, grid_y), method='linear') for i in range(5)]

plt.contourf(grid_x, grid_y, int_obs, levels = np.linspace(0,1))
plt.colorbar()

plt.contour(grid_x, grid_y, int_obs, levels = [0.05], colors = 'w')
for level in int_exp:
    plt.contour(grid_x, grid_y, level, levels = [0.05], colors = 'w', linestyles =
↪'dashed')

plt.scatter(X[:,0], X[:,1], c = yobs, vmin = 0, vmax = 1);
```



[12]:
```
sb.glue("number_2d_successpoints", len(X))
```

Data type cannot be displayed: application/papermill.record+json

5.6 Multibin Coupled HistoSys

[1]:
```
%pylab inline
```

```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
import logging
import json

import pyhf
from pyhf import Model

logging.basicConfig(level = logging.INFO)
```

[3]:
```
def prep_data(sourcedata):
    spec =  {
        'channels': [
            {
                'name': 'signal',
                'samples': [
                    {
                        'name': 'signal',
                        'data': sourcedata['signal']['bindata']['sig'],
                        'modifiers': [
                            {
                                'name': 'mu',
                                'type': 'normfactor',
                                'data': None
                            }
                        ]
```

(continues on next page)

(continued from previous page)

```
                    },
                    {
                        'name': 'bkg1',
                        'data': sourcedata['signal']['bindata']['bkg1'],
                        'modifiers': [
                            {
                                'name': 'coupled_histosys',
                                'type': 'histosys',
                                'data':  {'lo_data': sourcedata['signal']['bindata']['
↪'bkg1_dn'], 'hi_data': sourcedata['signal']['bindata']['bkg1_up']}
                            }
                        ]
                    },
                    {
                        'name': 'bkg2',
                        'data': sourcedata['signal']['bindata']['bkg2'],
                        'modifiers': [
                            {
                                'name': 'coupled_histosys',
                                'type': 'histosys',
                                'data':  {'lo_data': sourcedata['signal']['bindata']['
↪'bkg2_dn'], 'hi_data': sourcedata['signal']['bindata']['bkg2_up']}
                            }
                        ]
                    }
                ]
            },
            {
                'name': 'control',
                'samples': [
                    {
                        'name': 'background',
                        'data': sourcedata['control']['bindata']['bkg1'],
                        'modifiers': [
                            {
                                'name': 'coupled_histosys',
                                'type': 'histosys',
                                'data':  {'lo_data': sourcedata['control']['bindata']['
↪'bkg1_dn'], 'hi_data': sourcedata['control']['bindata']['bkg1_up']}
                            }
                        ]
                    }
                ]
            }
        ]
    }
    pdf  = Model(spec)
    data = []
    for c in pdf.spec['channels']:
        data += sourcedata[c['name']]['bindata']['data']
    data = data + pdf.config.auxdata
    return data, pdf
```

[4]:
```
validation_datadir = '../../validation/data'
```

[5]:
```
source = json.load(open(validation_datadir + '/2bin_2channel_coupledhisto.json'))

data, pdf = prep_data(source['channels'])

print(data)

init_pars = pdf.config.suggested_init()
par_bounds = pdf.config.suggested_bounds()

unconpars = pyhf.optimizer.unconstrained_bestfit(pyhf.utils.loglambdav, data, pdf,␣
↪init_pars, par_bounds)
print('parameters post unconstrained fit: {}'.format(unconpars))

conpars = pyhf.optimizer.constrained_bestfit(pyhf.utils.loglambdav, 0.0, data, pdf,␣
↪init_pars, par_bounds)
print('parameters post constrained fit: {}'.format(conpars))

pdf.expected_data(conpars)
```

```
INFO:pyhf.pdf:Validating spec against schema: /home/jovyan/pyhf/pyhf/data/spec.json
INFO:pyhf.pdf:adding modifier mu (1 new nuisance parameters)
INFO:pyhf.pdf:adding modifier coupled_histosys (1 new nuisance parameters)
```

```
[170.0, 220.0, 110.0, 105.0, 0.0]
parameters post unconstrained fit: [1.05563069e-12 4.00000334e+00]
parameters post constrained fit: [0.         4.00000146]
```

[5]:
```
array([ 1.25000007e+02,  1.60000022e+02,  2.10000022e+02, -8.00631284e-05,
        4.00000146e+00])
```

[6]:
```
def plot_results(test_mus, cls_obs, cls_exp, poi_tests, test_size = 0.05):
    plt.plot(poi_tests,cls_obs, c = 'k')
    for i,c in zip(range(5),['grey','grey','grey','grey','grey']):
        plt.plot(poi_tests, cls_exp[i], c = c)
    plt.plot(poi_tests,[test_size]*len(test_mus), c = 'r')
    plt.ylim(0,1)

def invert_interval(test_mus, cls_obs, cls_exp, test_size = 0.05):
    point05cross = {'exp':[],'obs':None}
    for cls_exp_sigma in cls_exp:
        yvals = cls_exp_sigma
        point05cross['exp'].append(np.interp(test_size,
                                             list(reversed(yvals)),
                                             list(reversed(test_mus))))

    yvals = cls_obs
    point05cross['obs'] = np.interp(test_size,
                                    list(reversed(yvals)),
                                    list(reversed(test_mus)))
    return point05cross


poi_tests = np.linspace(0, 5, 61)
tests = [pyhf.utils.hypotest(poi_test, data, pdf, init_pars, par_bounds, return_
↪expected_set=True)
         for poi_test in poi_tests]
cls_obs = np.array([test[0] for test in tests]).flatten()
cls_exp = [np.array([test[1][i] for test in tests]).flatten() for i in range(5)]
```

(continues on next page)

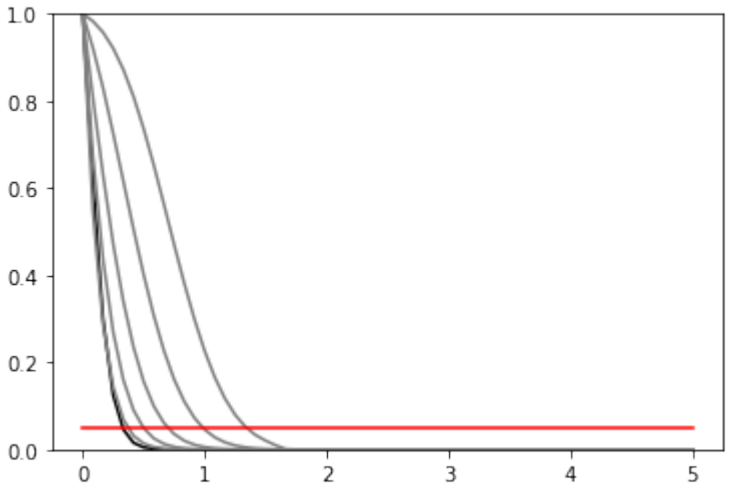
(continued from previous page)

```
print('\n')
plot_results(poi_tests, cls_obs, cls_exp, poi_tests)
invert_interval(poi_tests, cls_obs, cls_exp)
```

```
INFO:pyhf.pdf:Validating spec against schema: /home/jovyan/pyhf/pyhf/data/spec.json
INFO:pyhf.pdf:adding modifier mu (1 new nuisance parameters)
INFO:pyhf.pdf:adding modifier coupled_histosys (1 new nuisance parameters)
```

```
[170.0, 220.0, 110.0, 105.0, 0.0]
parameters post unconstrained fit: [1.05563069e-12 4.00000334e+00]
parameters post constrained fit: [0.         4.00000146]
```

[6]:
```
{'exp': [0.37566312243018246,
  0.49824494455027707,
  0.7023047842202288,
  0.9869744452422563,
  1.3443167343146392],
 'obs': 0.329179494864517}
```



CHAPTER

SIX

TALKS

We are always interested in talking about `pyhf`. See the abstract and a list of previously given presentations and feel free to invite us to your next conference/workshop/meeting!

6.1 Abstract

> The HistFactory p.d.f. template [CERN-OPEN-2012-016] is per-se independent of its implementation in ROOT and it is useful to be able to run statistical analysis outside of the ROOT, RooFit, RooStats framework. pyhf is a pure-python implementation of that statistical model for multi-bin histogram-based analysis and its interval estimation is based on the asymptotic formulas of "Asymptotic formulae for likelihood-based tests of new physics" [arxiv:1007.1727]. pyhf supports modern computational graph libraries such as TensorFlow and PyTorch in order to make use of features such as auto-differentiation and GPU acceleration.

```
The HistFactory p.d.f. template
\href{https://cds.cern.ch/record/1456844}{[CERN-OPEN-2012-016]} is
per-se independent of its implementation in ROOT and it is useful to be
able to run statistical analysis outside of the ROOT, RooFit, RooStats
framework. pyhf is a pure-python implementation of that statistical
model for multi-bin histogram-based analysis and its interval
estimation is based on the asymptotic formulas of "Asymptotic formulae
for likelihood-based tests of new physics"
\href{https://arxiv.org/abs/1007.1727}{[arxiv:1007.1727]}. pyhf
supports modern computational graph libraries such as TensorFlow and
PyTorch in order to make use of features such as autodifferentiation
and GPU acceleration.
```

6.2 Presentations

This list will be updated with talks given on `pyhf`:

- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: a pure Python implementation of HistFactory with tensors and autograd. DIANA Meeting - pyhf, October 2018. URL: https://indico.cern.ch/event/759480/.
- Matthew Feickert, Lukas Heinrich, Giordon Stark, and Kyle Cranmer. pyhf: pure-Python implementation of HistFactory models with autograd. (Internal) Joint Machine Learning & Statistics Fora Meeting, September 2018. URL: https://indico.cern.ch/event/757657/contributions/3141134/.
- Lukas Heinrich. Gaussian Process Shape Estimation and Systematics. (Internal) Joint Machine Learning & Statistics Fora Meeting, Dec 2018. URL: https://indico.cern.ch/event/777561/contributions/3234669/.

- Lukas Heinrich. pyhf: Full Run-2 ATLAS likelihoods. (Internal) Joint Machine Learning & Statistics Fora Meeting, May 2019. URL: https://indico.cern.ch/event/817483/contributions/3412907/.
- Lukas Heinrich, Matthew Feickert, Giordon Stark, and Kyle Cranmer. pyhf: A standalone HistFactory Implementation. (Re)interpreting the results of new physics searches at the LHC Workshop, May 2018. URL: https://indico.cern.ch/event/702612/contributions/2958658/.
- Giordon Stark. New techniques for use of public likelihoods for reinterpretation of search results. 27th International Conference on Supersymmetry and Unification of Fundamental Interactions (SUSY2019), May 2019. URL: https://indico.cern.ch/event/746178/contributions/3396797/.

6.3 Posters

This list will be updated with posters presented on `pyhf`:

- Lukas Heinrich, Matthew Feickert, Giordon Stark, and Kyle Cranmer. pyhf: auto-differentiable binned statistical models. 19th International Workshop on Advanced Computing and Analysis Techniques in Physics Research (ACAT 2019), March 2019. URL: https://indico.cern.ch/event/708041/contributions/3272095/.

CHAPTER

SEVEN

INSTALLATION

To install, we suggest first setting up a virtual environment

```
# Python3
python3 -m venv pyhf
```

```
# Python2
virtualenv --python=$(which python) pyhf
```

and activating it

```
source pyhf/bin/activate
```

7.1 Install latest stable release from PyPI...

7.1.1 ... with NumPy backend

```
pip install pyhf
```

7.1.2 ... with TensorFlow backend

```
pip install pyhf[tensorflow]
```

7.1.3 ... with PyTorch backend

```
pip install pyhf[torch]
```

7.1.4 ... with MXNet backend

```
pip install pyhf[mxnet]
```

7.1.5 ... with all backends

```
pip install pyhf[tensorflow,torch,mxnet]
```

7.1.6 ... with xml import/export functionality

```
pip install pyhf[xmlio]
```

7.2 Install latest development version from GitHub...

7.2.1 ... with NumPy backend

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git#egg=pyhf"
```

7.2.2 ... with TensorFlow backend

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git
↪#egg=pyhf[tensorflow]"
```

7.2.3 ... with PyTorch backend

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git
↪#egg=pyhf[torch]"
```

7.2.4 ... with MXNet backend

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git
↪#egg=pyhf[mxnet]"
```

7.2.5 ... with all backends

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git
↪#egg=pyhf[tensorflow,torch,mxnet]"
```

7.2.6 ... with xml import/export functionality

```
pip install --ignore-installed -U "git+https://github.com/diana-hep/pyhf.git
↪#egg=pyhf[xmlio]"
```

7.3 Updating `pyhf`

Rerun the installation command. As the upgrade flag, `-U`, is used then the libraries will be updated.

CHAPTER

EIGHT

DEVELOPING

To develop, we suggest using virtual environments together with `pip` or using pipenv. Once the environment is activated, clone the repo from GitHub

```
git clone https://github.com/diana-hep/pyhf.git
```

and install all necessary packages for development

```
pip install --ignore-installed -U -e .[complete]
```

Then setup the Git pre-commit hook for Black by running

```
pre-commit install
```

CHAPTER

NINE

FAQ

Frequently Asked Questions about `pyhf` and its use.

9.1 Questions

9.1.1 Is it possible to set the backend from the CLI?

Not at the moment. Pull Requests are welcome.

See also:

- #266

9.2 Troubleshooting

- `import torch` or `import pyhf` causes a `Segmentation fault (core dumped)`

  This is may be the result of a conflict with the NVIDIA drivers that you have installed on your machine. Try uninstalling and completely removing all of them from your machine

  ```
  # On Ubuntu/Debian
  sudo apt-get purge nvidia*
  ```

  and then installing the latest versions.

CHAPTER

TEN

API

10.1 Top-Level

default_backend	NumPy backend for pyhf
default_optimizer	
tensorlib	NumPy backend for pyhf
optimizer	
get_backend()	Get the current backend and the associated optimizer
set_backend(*args, **kwargs)	

10.1.1 pyhf.default_backend

pyhf.**default_backend = <pyhf.tensor.numpy_backend.numpy_backend object>**
NumPy backend for pyhf

10.1.2 pyhf.default_optimizer

pyhf.**default_optimizer = <pyhf.optimize.opt_scipy.scipy_optimizer object>**

10.1.3 pyhf.tensorlib

pyhf.**tensorlib = <pyhf.tensor.numpy_backend.numpy_backend object>**
NumPy backend for pyhf

10.1.4 pyhf.optimizer

pyhf.**optimizer = <pyhf.optimize.opt_scipy.scipy_optimizer object>**

10.1.5 pyhf.get_backend

pyhf.**get_backend**()
Get the current backend and the associated optimizer

Example

```
>>> import pyhf
>>> pyhf.get_backend()
(<pyhf.tensor.numpy_backend.numpy_backend object at 0x...>, <pyhf.optimize.opt_
↪scipy.scipy_optimizer object at 0x...>)
```

Returns backend, optimizer

10.1.6 pyhf.set_backend

pyhf.**set_backend**(**args*, ***kwargs*)

10.2 Making Probability Distribution Functions (PDFs)

Workspace	An object that is built from a JSON spec that follows *workspace.json*.
Model	
_ModelConfig	

10.2.1 Workspace

class pyhf.pdf.**Workspace**(*spec*, ***config_kwargs*)
Bases: object

An object that is built from a JSON spec that follows *workspace.json*.

Methods

__init__(*spec*, ***config_kwargs*)
Initialize self. See help(type(self)) for accurate signature.

data(*model*, *with_aux=True*)
Return the data for the supplied model with or without auxiliary data from the model.

The model is needed as the order of the data depends on the order of the channels in the model.

Parameters

- **model** – A model object adhering to the schema model.json
- **with_aux** – Whether to include auxiliary data from the model or not

Returns A list of numbers

Return type data

get_measurement(**config_kwargs)
Get (or create) a measurement object using the following logic:

1. if the poi name is given, create a measurement object for that poi
2. if the measurement name is given, find the measurement for the given name
3. if the measurement index is given, return the measurement at that index
4. if there are measurements but none of the above have been specified, return the 0th measurement
5. otherwise, raises *InvalidMeasurement*

Parameters

- **poi_name** – The name of the parameter of interest to create a new measurement from
- **measurement_name** – The name of the measurement to use
- **measurement_index** – The index of the measurement to use

Returns A measurement object adhering to the schema defs.json#/definitions/measurement

Return type measurement

model(**config_kwargs)
Create a model object with/without patches applied.

Parameters **patches** – A list of JSON patches to apply to the model specification

Returns A model object adhering to the schema model.json

Return type model

10.2.2 Model

class pyhf.pdf.**Model**(*spec*, **config_kwargs)
Bases: object

Methods

__init__(*spec*, **config_kwargs)
Initialize self. See help(type(self)) for accurate signature.

constraint_logpdf(*auxdata*, *pars*)

expected_actualdata(*pars*)
For a single channel single sample, we compute

Pois(d | fac(pars) * (delta(pars) + nom)) * Gaus(a | pars[is_gaus], sigmas) * Pois(a * cfac | pars[is_poi] * cfac)

where:

- delta(pars) is the result of an apply(pars) of combined modifiers with 'addition' op_code
- factor(pars) is the result of apply(pars) of combined modifiers with 'multiplication' op_code
- pars[is_gaus] are the subset of parameters that are constrained by gauss (with sigmas accordingly, some of which are computed by modifiers)

- pars[is_pois] are the poissons and their rates (they come with their own additional factors unrelated to factor(pars) which are also computed by the finalize() of the modifier)

So in the end we only make 3 calls to pdfs

1. The main pdf of data and modified rates
2. All Gaussian constraint as one call
3. All Poisson constraints as one call

expected_auxdata(*pars*)

expected_data(*pars*, *include_auxdata=True*)

logpdf(*pars*, *data*)

mainlogpdf(*maindata*, *pars*)

pdf(*pars*, *data*)

10.2.3 _ModelConfig

class pyhf.pdf.**_ModelConfig**(*spec*, ***config_kwargs*)
Bases: object

Methods

__init__(*spec*, ***config_kwargs*)
Initialize self. See help(type(self)) for accurate signature.

par_slice(*name*)

param_set(*name*)

set_poi(*name*)

suggested_bounds()

suggested_init()

10.3 Backends

The computational backends that pyhf provides interfacing for the vector-based calculations.

mxnet_backend.mxnet_backend	
numpy_backend.numpy_backend	NumPy backend for pyhf
pytorch_backend.pytorch_backend	
tensorflow_backend.tensorflow_backend	

10.3.1 numpy_backend

class pyhf.tensor.numpy_backend.**numpy_backend**(***kwargs*)
Bases: object

NumPy backend for pyhf

Methods

__init__(***kwargs*)
Initialize self. See help(type(self)) for accurate signature.

abs(*tensor*)

astensor(*tensor_in*, *dtype='float'*)
Convert to a NumPy array.

Parameters **tensor_in** (*Number or Tensor*) – Tensor object

Returns A multi-dimensional, fixed-size homogenous array.

Return type *numpy.ndarray*

boolean_mask(*tensor*, *mask*)

clip(*tensor_in*, *min_value*, *max_value*)
Clips (limits) the tensor values to be within a specified min and max.

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> a = pyhf.tensorlib.astensor([-2, -1, 0, 1, 2])
>>> pyhf.tensorlib.clip(a, -1, 1)
array([-1., -1.,  0.,  1.,  1.])
```

Parameters

- **tensor_in** (*tensor*) – The input tensor object
- **min_value** (*scalar* or *tensor* or *None*) – The minimum value to be cliped to
- **max_value** (*scalar* or *tensor* or *None*) – The maximum value to be cliped to

Returns A clipped *tensor*

Return type NumPy ndarray

concatenate(*sequence*, *axis=0*)
Join a sequence of arrays along an existing axis.

Parameters

- **sequence** – sequence of tensors
- **axis** – dimension along which to concatenate

Returns the concatenated tensor

Return type output

divide(*tensor_in_1*, *tensor_in_2*)

einsum(*subscripts*, **operands*)
Evaluates the Einstein summation convention on the operands.

Using the Einstein summation convention, many common multi-dimensional array operations can be represented in a simple fashion. This function provides a way to compute such summations. The best way to understand this function is to try the examples below, which show how many common NumPy functions can be implemented as calls to einsum.

Parameters

- **subscripts** – str, specifies the subscripts for summation
- **operands** – list of array_like, these are the tensors for the operation

Returns the calculation based on the Einstein summation convention

Return type tensor

exp(*tensor_in*)

gather(*tensor*, *indices*)

isfinite(*tensor*)

log(*tensor_in*)

normal(*x*, *mu*, *sigma*)
The probability density function of the Normal distribution evaluated at `x` given parameters of mean of `mu` and standard deviation of `sigma`.

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> pyhf.tensorlib.normal(0.5, 0., 1.)
0.3520653267642995
```

Parameters

- **x** (*tensor* or *float*) – The value at which to evaluate the Normal distribution p.d.f.
- **mu** (*tensor* or *float*) – The mean of the Normal distribution
- **sigma** (*tensor* or *float*) – The standard deviation of the Normal distribution

Returns Value of Normal(x|mu, sigma)

Return type NumPy float

normal_cdf(*x*, *mu=0*, *sigma=1*)
The cumulative distribution function for the Normal distribution

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> pyhf.tensorlib.normal_cdf(0.8)
0.7881446014166034
```

Parameters

- **x** (*tensor* or *float*) – The observed value of the random variable to evaluate the CDF for
- **mu** (*tensor* or *float*) – The mean of the Normal distribution
- **sigma** (*tensor* or *float*) – The standard deviation of the Normal distribution

Returns The CDF

Return type NumPy float

normal_logpdf(*x*, *mu*, *sigma*)

ones(*shape*)

outer(*tensor_in_1*, *tensor_in_2*)

poisson(*n*, *lam*)
The continous approximation, using $n! = \Gamma(n+1)$, to the probability mass function of the Poisson distribution evaluated at `n` given the parameter `lam`.

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> pyhf.tensorlib.poisson(5., 6.)
0.16062314104797995
```

Parameters

- **n** (*tensor* or *float*) – The value at which to evaluate the approximation to the Poisson distribution p.m.f. (the observed number of events)
- **lam** (*tensor* or *float*) – The mean of the Poisson distribution p.m.f. (the expected number of events)

Returns Value of the continous approximation to Poisson(nllam)

Return type NumPy float

poisson_logpdf(*n*, *lam*)

power(*tensor_in_1*, *tensor_in_2*)

product(*tensor_in*, *axis=None*)

reshape(*tensor*, *newshape*)

shape(*tensor*)

simple_broadcast(**args*)
Broadcast a sequence of 1 dimensional arrays.

Example

```
>>> import pyhf
>>> pyhf.set_backend(pyhf.tensor.numpy_backend())
>>> pyhf.tensorlib.simple_broadcast(
...   pyhf.tensorlib.astensor([1]),
...   pyhf.tensorlib.astensor([2, 3, 4]),
...   pyhf.tensorlib.astensor([5, 6, 7]))
[array([1., 1., 1.]), array([2., 3., 4.]), array([5., 6., 7.])]
```

Parameters **args** (*Array of Tensors*) – Sequence of arrays

Returns The sequence broadcast together.

Return type list of Tensors

sqrt(*tensor_in*)

stack(*sequence*, *axis=0*)

sum(*tensor_in*, *axis=None*)

tolist(*tensor_in*)

where(*mask*, *tensor_in_1*, *tensor_in_2*)

zeros(*shape*)

10.4 Optimizers

opt_pytorch.pytorch_optimizer
opt_scipy.scipy_optimizer
opt_tflow.tflow_optimizer
opt_minuit.minuit_optimizer

10.4.1 scipy_optimizer

class pyhf.optimize.opt_scipy.**scipy_optimizer**(***kwargs*)

Bases: object

Methods

__init__(***kwargs*)

Initialize self. See help(type(self)) for accurate signature.

constrained_bestfit(*objective*, *constrained_mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

unconstrained_bestfit(*objective*, *data*, *pdf*, *init_pars*, *par_bounds*)

10.5 Modifiers

10.5.1 histosys

class pyhf.modifiers.**histosys**
Bases: object

Attributes

is_constrained = True

op_code = 'addition'

pdf_type = 'normal'

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.5.2 normfactor

class pyhf.modifiers.**normfactor**
Bases: object

Attributes

is_constrained = False

op_code = 'multiplication'

pdf_type = None

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.5.3 normsys

class pyhf.modifiers.**normsys**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'normal'

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.5.4 shapefactor

class pyhf.modifiers.**shapefactor**
Bases: object

Attributes

is_constrained = False

op_code = 'multiplication'

pdf_type = None

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.5.5 shapesys

class pyhf.modifiers.**shapesys**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'poisson'

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.5.6 staterror

class pyhf.modifiers.**staterror**
Bases: object

Attributes

is_constrained = True

op_code = 'multiplication'

pdf_type = 'normal'

Methods

__init__()
Initialize self. See help(type(self)) for accurate signature.

classmethod required_parset(*n_parameters*)

10.6 Interpolators

code0	The piecewise-linear interpolation strategy.
code1	The piecewise-exponential interpolation strategy.
code2	The quadratic interpolation and linear extrapolation strategy.
code4	The polynomial interpolation and exponential extrapolation strategy.
code4p	The piecewise-linear interpolation strategy, with polynomial at $\|a\| < 1$

10.6.1 code0

class pyhf.interpolators.**code0**(*histogramssets*, *subscribe=True*)
Bases: object

The piecewise-linear interpolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) + \underbrace{\sum_{p\in\text{Syst}} I_{\text{lin.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb})}_{\text{deltas to calculate}}$$

with

$$I_{\text{lin.}}(\alpha; I^0, I^+, I^-) = \begin{cases} \alpha(I^+ - I^0) & \alpha \geq 0 \\ \alpha(I^0 - I^-) & \alpha < 0 \end{cases}$$

Methods

__init__(*histogramssets*, *subscribe=True*)
Initialize self. See help(type(self)) for accurate signature.

10.6.2 code1

class pyhf.interpolators.**code1**(*histogramssets*, *subscribe=True*)
Bases: object

The piecewise-exponential interpolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) \underbrace{\prod_{p\in\text{Syst}} I_{\text{exp.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb})}_{\text{factors to calculate}}$$

with

$$I_{\text{exp.}}(\alpha; I^0, I^+, I^-) = \begin{cases} \left(\frac{I^+}{I^0}\right)^{\alpha} & \alpha \geq 0 \\ \left(\frac{I^-}{I^0}\right)^{-\alpha} & \alpha < 0 \end{cases}$$

Methods

__init__(*histogramssets*, *subscribe=True*)
Initialize self. See help(type(self)) for accurate signature.

10.6.3 code2

class pyhf.interpolators.**code2**(*histogramssets*, *subscribe=True*)
Bases: object

The quadratic interpolation and linear extrapolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) + \underbrace{\sum_{p\in\text{Syst}} I_{\text{quad.llin.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb})}_{\text{deltas to calculate}}$$

with

$$I_{\text{quad.|lin.}}(\alpha; I^0, I^+, I^-) = \begin{cases} (b+2a)(\alpha-1) & \alpha \geq 1 \\ a\alpha^2 + b\alpha & |\alpha| < 1 \\ (b-2a)(\alpha+1) & \alpha < -1 \end{cases}$$

and

$$a = \frac{1}{2}(I^+ + I^-) - I^0 \qquad \text{and} \qquad b = \frac{1}{2}(I^+ - I^-)$$

Methods

__init__ (*histogramssets*, *subscribe=True*)
Initialize self. See help(type(self)) for accurate signature.

10.6.4 code4

class pyhf.interpolators.**code4** (*histogramssets*, *subscribe=True*, *alpha0=1*)
Bases: object

The polynomial interpolation and exponential extrapolation strategy.

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) \underbrace{\prod_{p \in \text{Syst}} I_{\text{polylexp.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb}, \alpha_0)}_{\text{factors to calculate}}$$

with

$$I_{\text{polylexp.}}(\alpha; I^0, I^+, I^-, \alpha_0) = \begin{cases} \left(\frac{I^+}{I^0}\right)^{\alpha} & \alpha \geq \alpha_0 \\ 1 + \sum_{i=1}^{6} a_i \alpha^i & |\alpha| < \alpha_0 \\ \left(\frac{I^-}{I^0}\right)^{-\alpha} & \alpha < -\alpha_0 \end{cases}$$

and the a_i are fixed by the boundary conditions

$$\sigma_{sb}(\alpha = \pm\alpha_0), \left.\frac{\mathrm{d}\sigma_{sb}}{\mathrm{d}\alpha}\right|_{\alpha=\pm\alpha_0}, \text{and } \left.\frac{\mathrm{d}^2\sigma_{sb}}{\mathrm{d}\alpha^2}\right|_{\alpha=\pm\alpha_0}.$$

Namely that $\sigma_{sb}(\vec{\alpha})$ is continuous, and its first- and second-order derivatives are continuous as well.

Methods

__init__ (*histogramssets*, *subscribe=True*, *alpha0=1*)
Initialize self. See help(type(self)) for accurate signature.

10.6.5 code4p

class pyhf.interpolators.**code4p** (*histogramssets*, *subscribe=True*)
Bases: object

The piecewise-linear interpolation strategy, with polynomial at $|a| < 1$

$$\sigma_{sb}(\vec{\alpha}) = \sigma^0_{sb}(\vec{\alpha}) + \underbrace{\sum_{p \in \text{Syst}} I_{\text{lin.}}(\alpha_p; \sigma^0_{sb}, \sigma^+_{psb}, \sigma^-_{psb})}_{\text{deltas to calculate}}$$

Methods

__init__ (*histogramssets*, *subscribe=True*)
Initialize self. See help(type(self)) for accurate signature.

10.7 Exceptions

Various exceptions, apart from standard python exceptions, that are raised from using the `pyhf` API.

InvalidInterpCode	InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.
InvalidModifier	InvalidModifier is raised when an invalid modifier is requested.

10.7.1 InvalidInterpCode

class pyhf.exceptions.**InvalidInterpCode**
Bases: `Exception`

InvalidInterpCode is raised when an invalid/unimplemented interpolation code is requested.

10.7.2 InvalidModifier

class pyhf.exceptions.**InvalidModifier**
Bases: `Exception`

InvalidModifier is raised when an invalid modifier is requested. This includes:

- creating a custom modifier with the wrong structure
- initializing a modifier that does not exist, or has not been loaded

10.8 Utilities

generate_asimov_data(asimov_mu, data, pdf, ...)	
loglambdav(pars, data, pdf)	
pvals_from_teststat(sqrtqmu_v, sqrtqmuA_v[, ...])	The p-values for signal strength μ and Asimov strength μ' as defined in Equations (59) and (57) of **`arXiv:1007.1727`_**
pvals_from_teststat_expected(sqrtqmuA_v[, ...])	Computes the expected p-values CLsb, CLb and CLs for data corresponding to a given percentile of the alternate hypothesis.
qmu(mu, data, pdf, init_pars, par_bounds)	The test statistic, q_μ, for establishing an upper limit on the strength parameter, μ, as defiend in Equation (14) in **`arXiv:1007.1727`_** .
hypotest(poi_test, data, pdf[, init_pars, ...])	Computes p-values and test statistics for a single value of the parameter of interest

10.8.1 pyhf.utils.generate_asimov_data

pyhf.utils.**generate_asimov_data**(*asimov_mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

10.8.2 pyhf.utils.loglambdav

pyhf.utils.**loglambdav**(*pars*, *data*, *pdf*)

10.8.3 pyhf.utils.pvals_from_teststat

pyhf.utils.**pvals_from_teststat**(*sqrtqmu_v*, *sqrtqmuA_v*, *qtilde=False*)

The p-values for signal strength μ and Asimov strength μ' as defined in Equations (59) and (57) of arXiv:1007.1727

$$p_\mu = 1 - F\left(q_\mu|\mu'\right) = 1 - \Phi\left(q_\mu - \frac{(\mu - \mu')}{\sigma}\right)$$

with Equation (29)

$$\frac{(\mu - \mu')}{\sigma} = \sqrt{\Lambda} = \sqrt{q_{\mu,A}}$$

given the observed test statistics q_μ and $q_{\mu,A}$.

Parameters

- **sqrtqmu_v** (*Number or Tensor*) – The root of the calculated test statistic, $\sqrt{q_\mu}$
- **sqrtqmuA_v** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **qtilde** (*Bool*) – When `True` perform the calculation using the alternative test statistic, $\tilde{q}$, as defined in Equation (62) of arXiv:1007.1727

Returns The p-values for the signal + background, background only, and signal only hypotheses respectivley

Return type Tuple of Floats

10.8.4 pyhf.utils.pvals_from_teststat_expected

pyhf.utils.**pvals_from_teststat_expected**(*sqrtqmuA_v*, *nsigma=0*)

Computes the expected p-values CLsb, CLb and CLs for data corresponding to a given percentile of the alternate hypothesis.

Parameters

- **sqrtqmuA_v** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **nsigma** (*Number or Tensor*) – The number of standard deviations of variations of the signal strength from the background only hypothesis ($\mu = 0$)

Returns The p-values for the signal + background, background only, and signal only hypotheses respectivley

Return type Tuple of Floats

10.8.5 pyhf.utils.qmu

pyhf.utils.**qmu**(*mu*, *data*, *pdf*, *init_pars*, *par_bounds*)

The test statistic, q_μ, for establishing an upper limit on the strength parameter, μ, as defiend in Equation (14) in arXiv:1007.1727 .

$$q_\mu = \begin{cases} -2\ln\lambda\left(\mu\right), & \hat{\mu} < \mu, \\ 0, & \hat{\mu} > \mu \end{cases} \tag{10.1}$$

Parameters

- **mu** (*Number or Tensor*) – The signal strength parameter
- **data** (*Tensor*) – The data to be considered
- **pdf** (pyhf.pdf.Model) – The HistFactory statistical model used in the likelihood ratio calculation
- **init_pars** (*Tensor*) – The initial parameters
- **par_bounds** (*Tensor*) – The bounds on the paramter values

Returns The calculated test statistic, q_μ

Return type Float

10.8.6 pyhf.utils.hypotest

pyhf.utils.**hypotest**(*poi_test*, *data*, *pdf*, *init_pars=None*, *par_bounds=None*, *qtilde=False*, ***kwargs*)

Computes p-values and test statistics for a single value of the parameter of interest

Parameters

- **poi_test** (*Number or Tensor*) – The value of the parameter of interest (POI)
- **data** (*Number or Tensor*) – The root of the calculated test statistic given the Asimov data, $\sqrt{q_{\mu,A}}$
- **pdf** (pyhf.pdf.Model) – The HistFactory statistical model
- **init_pars** (*Array or Tensor*) – The initial parameter values to be used for minimization
- **par_bounds** (*Array or Tensor*) – The parameter value bounds to be used for minimization
- **qtilde** (*Bool*) – When `True` perform the calculation using the alternative test statistic, $\tilde{q}$, as defined in Equation (62) of arXiv:1007.1727

Keyword Arguments

- **return_tail_probs** (*bool*) – Bool for returning CL_{s+b} and CL_b
- **return_expected** (*bool*) – Bool for returning $\mathrm{CL}_{\mathrm{exp}}$
- **return_expected_set** (*bool*) – Bool for returning the $(-2,-1,0,1,2)\sigma$ $\mathrm{CL}_{\mathrm{exp}}$ — the "Brazil band"
- **return_test_statistics** (*bool*) – Bool for returning q_μ and $q_{\mu,A}$

Returns

- CL_s: The p-value compared to the given threshold α, typically taken to be 0.05, defined in arXiv:1007.1727 as

$$\mathrm{CL}_s = \frac{\mathrm{CL}_{s+b}}{\mathrm{CL}_b} = \frac{p_{s+b}}{1-p_b}$$

to protect against excluding signal models in which there is little sensitivity. In the case that $\mathrm{CL}_s \leq \alpha$ the given signal model is excluded.

- $[\mathrm{CL}_{s+b}, \mathrm{CL}_b]$: The signal + background p-value and 1 minus the background only p-value as defined in Equations (75) and (76) of arXiv:1007.1727

$$\mathrm{CL}_{s+b} = p_{s+b} = \int_{q_\mathrm{obs}}^{\infty} f\left(q\,|s+b\right)\,dq = 1 - \Phi\left(\frac{q_\mathrm{obs} + 1/\sigma_{s+b}^2}{2/\sigma_{s+b}}\right)$$

$$\mathrm{CL}_b = 1 - p_b = 1 - \int_{-\infty}^{q_\mathrm{obs}} f\left(q\,|b\right)\,dq = 1 - \Phi\left(\frac{q_\mathrm{obs} - 1/\sigma_{b}^2}{2/\sigma_{b}}\right)$$

with Equations (73) and (74) for the mean

$$E\left[q\right] = \frac{1-2\mu}{\sigma^2}$$

and variance

$$V\left[q\right] = \frac{4}{\sigma^2}$$

of the test statistic q under the background only and and signal + background hypotheses. Only returned when `return_tail_probs` is `True`.

- $\mathrm{CL}_{s,\mathrm{exp}}$: The expected CL_s value corresponding to the test statistic under the background only hypothesis $(\mu = 0)$. Only returned when `return_expected` is `True`.
- $\mathrm{CL}_{s,\mathrm{exp}}$ band: The set of expected CL_s values corresponding to the median significance of variations of the signal strength from the background only hypothesis $(\mu = 0)$ at $(-2, -1, 0, 1, 2)\sigma$. That is, the p-values that satisfy Equation (89) of arXiv:1007.1727

$$\mathrm{band}_{N\sigma} = \mu' + \sigma\,\Phi^{-1}\left(1-\alpha\right) \pm N\sigma$$

for $\mu' = 0$ and $N \in \{-2, -1, 0, 1, 2\}$. These values define the boundaries of an uncertainty band sometimes referred to as the "Brazil band". Only returned when `return_expected_set` is `True`.

- $[q_\mu, q_{\mu,A}]$: The test statistics for the observed and Asimov datasets respectively. Only returned when `return_test_statistics` is `True`.

Return type Tuple of Floats and lists of Floats

CHAPTER

ELEVEN

USE AND CITATIONS

Updating list of citations and use cases of `pyhf`:

- Lukas Heinrich, Holger Schulz, Jessica Turner, and Ye-Ling Zhou. Constraining A_4 Leptonic Flavour Model Parameters at Colliders and Beyond. 2018. arXiv:1810.05648.

CHAPTER

TWELVE

PURE-PYTHON FITTING/LIMIT-SETTING/INTERVAL ESTIMATION HISTFACTORY-STYLE

The HistFactory p.d.f. template [CERN-OPEN-2012-016] is per-se independent of its implementation in ROOT and sometimes, it's useful to be able to run statistical analysis outside of ROOT, RooFit, RooStats framework.

This repo is a pure-python implementation of that statistical model for multi-bin histogram-based analysis and its interval estimation is based on the asymptotic formulas of "Asymptotic formulae for likelihood-based tests of new physics" [arxiv:1007.1727]. The aim is also to support modern computational graph libraries such as PyTorch and TensorFlow in order to make use of features such as autodifferentiation and GPU acceleration.

12.1 Hello World

```
>>> import pyhf
>>> pdf = pyhf.simplemodels.hepdata_like(signal_data=[12.0, 11.0], bkg_data=[50.0, 52.
↪0], bkg_uncerts=[3.0, 7.0])
>>> CLs_obs, CLs_exp = pyhf.utils.hypotest(1.0, [51, 48] + pdf.config.auxdata, pdf,␣
↪return_expected=True)
>>> print('Observed: {}, Expected: {}'.format(CLs_obs, CLs_exp))
Observed: [0.05290116], Expected: [0.06445521]
```

12.2 What does it support

Implemented variations:

- [x] HistoSys
- [x] OverallSys
- [x] ShapeSys
- [x] NormFactor
- [x] Multiple Channels
- [x] Import from XML + ROOT via uproot
- [x] ShapeFactor
- [x] StatError
- [x] Lumi Uncertainty

Computational Backends:

- [x] NumPy
- [x] PyTorch
- [x] TensorFlow
- [x] MXNet

Available Optimizers

NumPy	Tensorflow	PyTorch	MxNet
SLSQP (`scipy.optimize`)	Newton's Method (autodiff)	Newton's Method (autodiff)	N/A
MINUIT (`iminuit`)	.	.	.

12.3 Todo

- [ ] StatConfig
- [ ] Non-asymptotic calculators

results obtained from this package are validated against output computed from HistFactory workspaces

12.4 A one bin example

```
nobs = 55, b = 50, db = 7, nom_sig = 10.
```

12.5 A two bin example

```
bin 1: nobs = 100, b = 100, db = 15., nom_sig = 30.
bin 2: nobs = 145, b = 150, db = 20., nom_sig = 45.
```

12.6 Installation

To install `pyhf` from PyPI with the NumPy backend run

```
pip install pyhf
```

and to install `pyhf` with additional backends run

```
pip install pyhf[tensorflow,torch,mxnet]
```

or a subset of the options.

To uninstall run

```
pip uninstall pyhf
```

12.7 Authors

Please check the contribution statistics for a list of contributors

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[1] Glen Cowan, Kyle Cranmer, Eilam Gross, and Ofer Vitells. Asymptotic formulae for likelihood-based tests of new physics. *Eur. Phys. J. C*, 71:1554, 2011. arXiv:1007.1727, doi:10.1140/epjc/s10052-011-1554-0.

[2] Kyle Cranmer, George Lewis, Lorenzo Moneta, Akira Shibata, and Wouter Verkerke. HistFactory: A tool for creating statistical models for use with RooFit and RooStats. Technical Report CERN-OPEN-2012-016, New York U., New York, Jan 2012. URL: https://cds.cern.ch/record/1456844.

[3] Eamonn Maguire, Lukas Heinrich, and Graeme Watt. HEPData: a repository for high energy physics data. *J. Phys. Conf. Ser.*, 898(10):102006, 2017. arXiv:1704.05473, doi:10.1088/1742-6596/898/10/102006.

[4] ATLAS Collaboration. Measurements of Higgs boson production and couplings in diboson final states with the ATLAS detector at the LHC. *Phys. Lett. B*, 726:88, 2013. arXiv:1307.1427, doi:10.1016/j.physletb.2014.05.011.

[5] ATLAS Collaboration. Search for supersymmetry in final states with missing transverse momentum and multiple \(b\)-jets in proton–proton collisions at \(\sqrt s = 13\) \(\TeV \) with the ATLAS detector. ATLAS-CONF-2018-041, 2018. URL: https://cds.cern.ch/record/2632347.

[1] Histfactory definitions schema. Accessed: 2019-06-20. URL: https://diana-hep.org/pyhf/schemas/1.0.0/defs.json.

[2] Kyle Cranmer, George Lewis, Lorenzo Moneta, Akira Shibata, and Wouter Verkerke. HistFactory: A tool for creating statistical models for use with RooFit and RooStats. Technical Report CERN-OPEN-2012-016, New York U., New York, Jan 2012. URL: https://cds.cern.ch/record/1456844.

INDEX

Symbols

A

B

C

D

E

G

H

I

T

U

W

Z