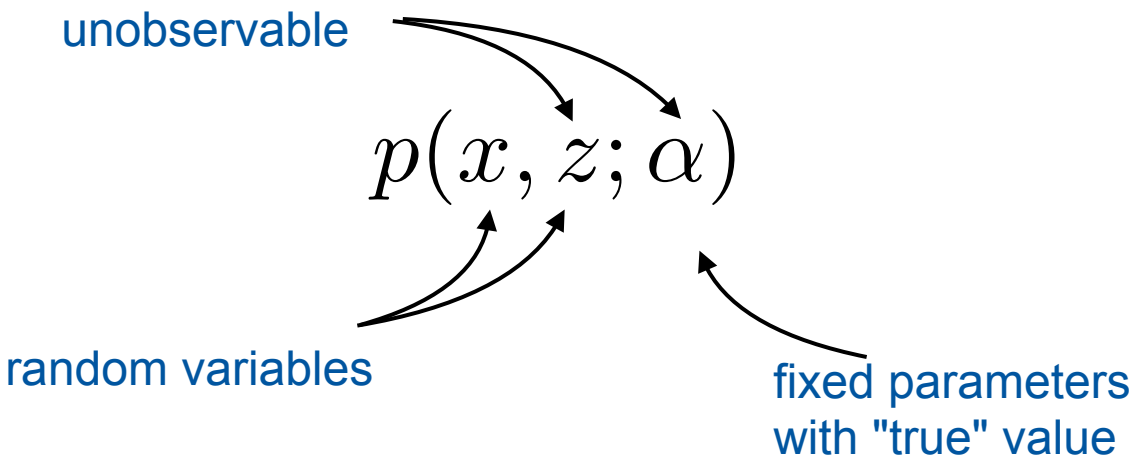
stats tools w/ ML tools

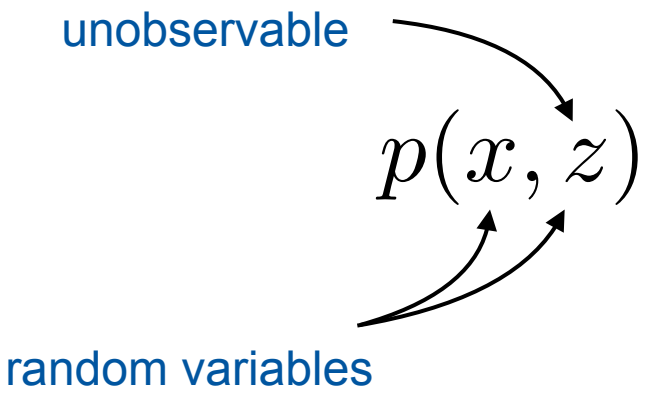
Lukas Heinrich, CERN

Rememer yesterday:

A lot of statistical modelling is about finding a representation of the measurement:

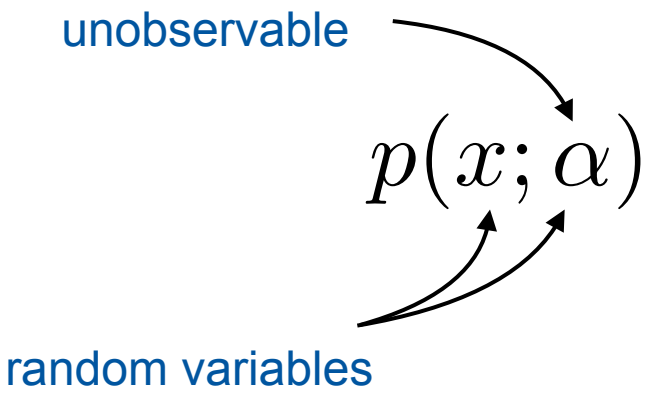


In "pure" bayesian inference, all interesting unobserved values are in the latent random variables



Inference $$p(z|x) = \frac{p(x|z)}{p(x)} p(z)$$

In "pure" frequentist inference, all interesting unobserved values are in the "fixed parameters"



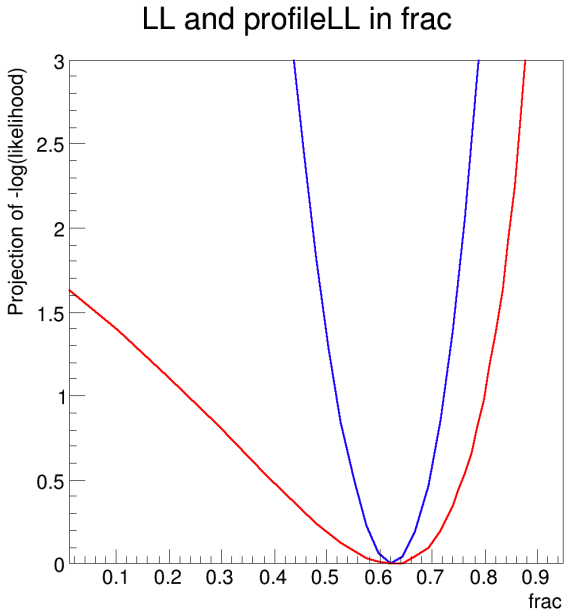
Inference (e.g. max likelihood) $\hat{\alpha}(x) = \mathrm{argmax}_{\alpha}\, p(x;\alpha)$

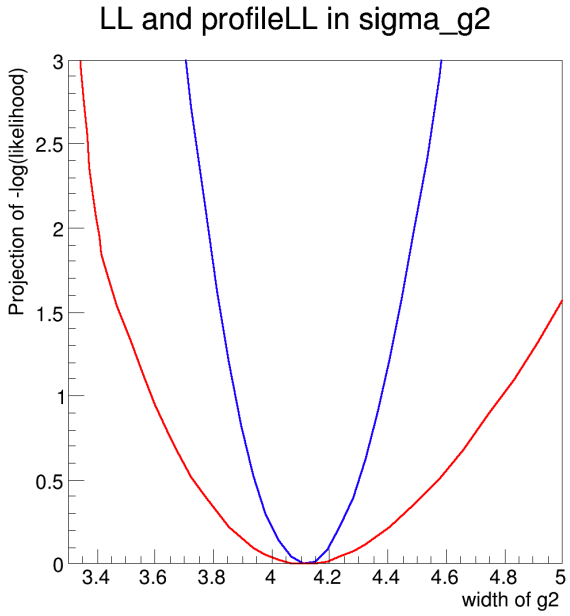
In HEP, a lot of inference is done in a frequentist framework:

$$\hat{\alpha}(x) = \text{argmax}_{\alpha}\ p(x;\alpha)$$

Likelihood Fitting is our bread and butter.

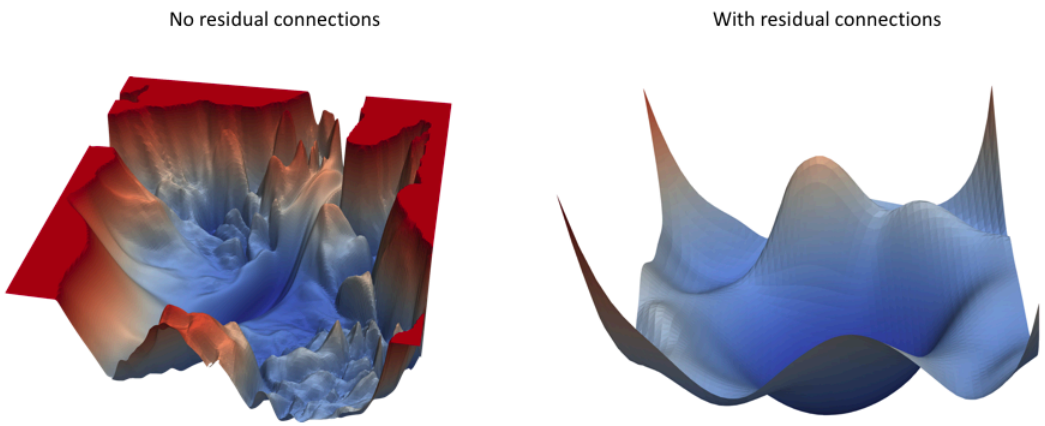
- **ML estimators objective: minimize negative log likelihood**





How does the ML community and help ?
minimization is their bread and butter, too!

- Minimization of "loss function" $L(\phi)$
- ML is more than "glorified fitting", similar enough



visualization of loss landscape

Minimization: derivatives are important!

In HEP:

- **e.g. MIGRAD uses gradients / hessians during minimization**

in ML:

- **Stochastic Gradient Descent (SGD) and other optimizers use gradients.**

$$\text{Repeat Until Convergence } \{$$

$$\omega \leftarrow \omega - \alpha * \nabla_w \sum_{1}^{m} L_m(w)$$

$$\}$$

Standard MIGRAD computes derivatives as "finite differences"

Finite difference in several variables [edit]

Finite differences can be considered in more than one variable. They are analogous to partial derivatives in several variables.

Some partial derivative approximations are:

$$f_x(x,y) \approx \frac{f(x+h,y)-f(x-h,y)}{2h}$$
$$f_y(x,y) \approx \frac{f(x,y+k)-f(x,y-k)}{2k}$$
$$f_{xx}(x,y) \approx \frac{f(x+h,y)-2f(x,y)+f(x-h,y)}{h^2}$$
$$f_{yy}(x,y) \approx \frac{f(x,y+k)-2f(x,y)+f(x,y-k)}{k^2}$$
$$f_{xy}(x,y) \approx \frac{f(x+h,y+k)-f(x+h,y-k)-f(x-h,y+k)+f(x-h,y-k)}{4hk}.$$

when they are supplied by the user. Accurate numerical differentiation is difficult and takes cpu time, because the function value has to be determined several times for a single derivative, and the derivative value can be affected by round-off errors. For large scale optimization problems analytical derivatives seem to be essential. MINUIT performs a lot of checks during minimization and is able to detect bugs in the user code.
There was a limit of 50 (variable parameters) in the F77 version. The C++ version has no limit

V. Blobel

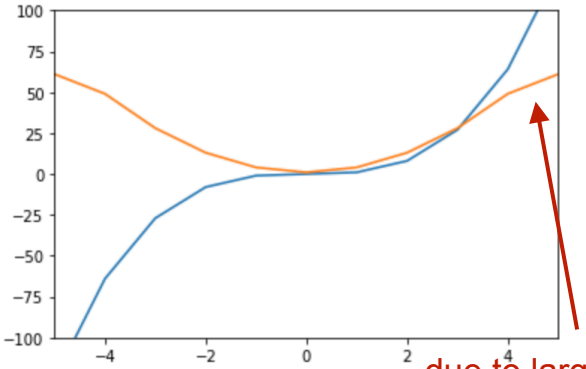
Number of function evaluations grows exponentially
Remembter: ML can have BILLIONS of parameters.. how do they do it?

Automatic Differentiation (AutoDiff) (separate lecture)

In short: a way to automatically compute derivatives w.r.t to many variables to Machine Precision.

Backpropagation: Applied AutoDiff

```
In [3]: x = np.linspace(-5,5,11)
        y = x**3
        g = np.gradient(y,x)
        plt.plot(x,y)
        plt.plot(x,g)
        plt.xlim(-5,5)
        plt.ylim(-100,100)

Out[3]: (-100, 100)
```

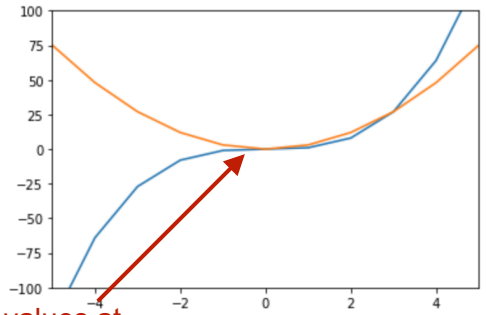


due to large step size approximation is bad :(

Using Finite Differences

```
In [3]: import jax
        import matplotlib.pyplot as plt
        import jax.numpy as np
        from jax import vmap

        %matplotlib inline

In [5]: def func(x):
            return np.power(x,3)

        gradient = jax.grad(func)

        x = np.linspace(-5,5,11)
        y = func(x)
        g = vmap(gradient)(x)

        plt.plot(x,y)
        plt.plot(x,g)
        plt.xlim(-5,5)
        plt.ylim(-100,100)

Out[5]: (-100, 100)
```

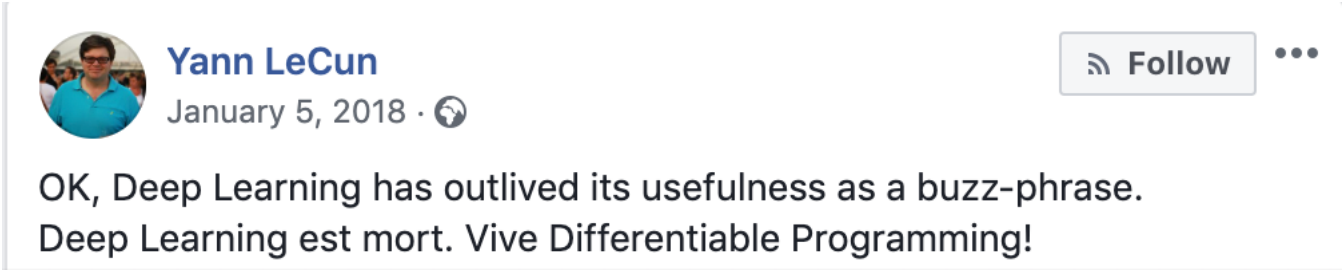
using ML tools



exact values at evaluted points for derivative

Using AutoDiff

Machine Learning as Differentiable Programming



In some sense, all of ML is about writing programs in programming languages that make computing gradients easy using autodiff

"Programming languages" in this case:

TensorFlow

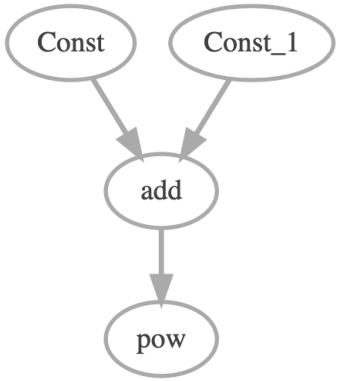
PyTorch

A differentiable program only uses a subets of numeric compuation primitives

- builds up a graph of fundamental differentiable operations
  - sums, sqrt, log, power, ...
- autodiff engine can use it to efficiently compute derivatives

```
In [1]: import tensorflow as tf
        import tfgraphviz as tfg

In [2]: intermediate = tf.constant(2) + tf.constant(3)
        final = intermediate ** 2

In [3]: g = tfg.board(tf.get_default_graph())
        g
```

Out[3]:



Differentiable Programming Language

- set of diff'able primitive operations
- autodiff engine

Numpy API: the standard numeric tensor-manipulation API

All ML tools more or less implement a similar API to deal with multi-dimensional arrays, very much inspired by `numpy`

```
In [23]: import tensorflow as tf
         import numpy as np
         import jax.numpy as jnp
         import torch

In [12]: a = np.array([1,2,3]) + np.array([4,5,6])
         b = np.power(a,2)
         b

Out[12]: array([25, 49, 81])

In [15]: a = torch.Tensor([1,2,3]) + torch.Tensor([4,5,6])
         b = torch.pow(a,2)
         b

Out[15]: tensor([25., 49., 81.])

In [25]: a = tf.constant([1,2,3]) +  tf.constant([4,5,6])
         b = tf.pow(a,2)
         tf.Session().run(b)

Out[25]: array([25, 49, 81], dtype=int32)

In [26]: a = jnp.array([1,2,3]) + jnp.array([4,5,6])
         b = jnp.power(a,2)
         b

Out[26]: DeviceArray([25, 49, 81], dtype=int32)
```

Upshot:

if we want to use autodiff in our minimization, we need to write our models using numpy-like primitives

- **once we have that it's easy to port it to ML frameworks**

Ancilarry benefits:

- **once it's implemented in these new languages, get e.g. hardware acceleration (GPU) for free**
- **basically means implementing statistical models in python**
  - **user-friendly**

In the last year or so a few projects have emerged to re-implement standard HEP statistical models to ML-style differentiable programs:

pyhf: pure-python HistFactory (ATLAS, binned)

- tensorflow, pytorch, numpy, (jax) ..

https://github.com/diana-hep/pyhf/

zfit: unbinned modeling library

- tensorflow only

https://github.com/zfit/zfit

Combine on Tensorflow: (CMS, binned)

- tensorflow only

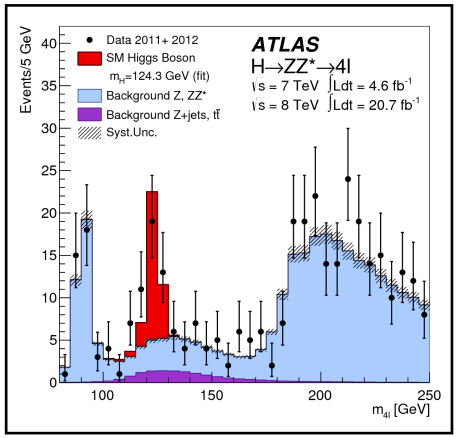
https://github.com/bendavid/HiggsAnalysis-CombinedLimit/tree/tensorflowfit

Example: pyhf

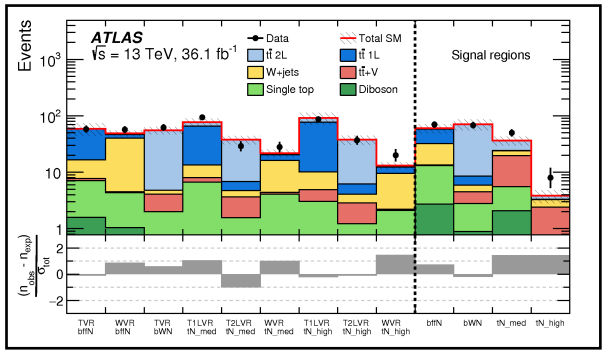
HistFactory is a very widely used tool to do template-based binned analysis for the LHC (similar to combine)

A lot of results we publish in ATLAS (CMS) use HistFactory (Combine) for binned analysis

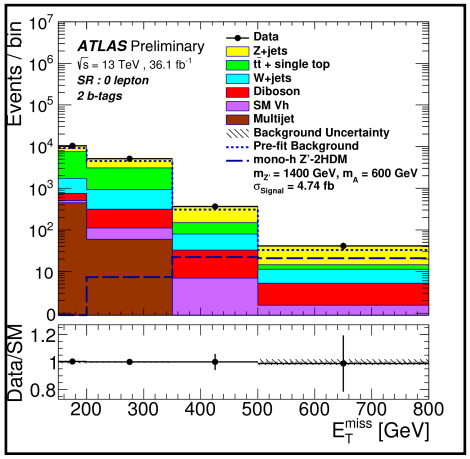
SM



SUSY



Exotics



HistFactory: a single simple formula involving Poissons, Gaussians powering a large percentag of ATLAS stats

HistFactory: A tool for creating statistical models for use with RooFit and RooStats

Kyle Cranmer, George Lewis, Lorenzo Moneta, Akira Shibata, Wouter Verkerke

June 20, 2012

Contents

- **simultaneous measurement of multiple,binned phase-phase regions**
- **(frequentist) constaint terms for nuisance parameters**

$$f(\boldsymbol{n}, \boldsymbol{a} \mid \boldsymbol{\eta}, \boldsymbol{\chi}) = \underbrace{\prod_{c \in \text{channels}} \prod_{b \in \text{bins}_c} \text{Pois}\left(n_{cb} \mid \nu_{cb}\left(\boldsymbol{\eta}, \boldsymbol{\chi}\right)\right)}_{\text{Simultaneous measurement of multiple channels}} \underbrace{\prod_{\chi \in \boldsymbol{\chi}} c_{\chi}(a_{\chi} \mid \chi)}_{\text{constraint terms for “auxiliary measurements”}},$$

Main Measurement **Constraint Terms**

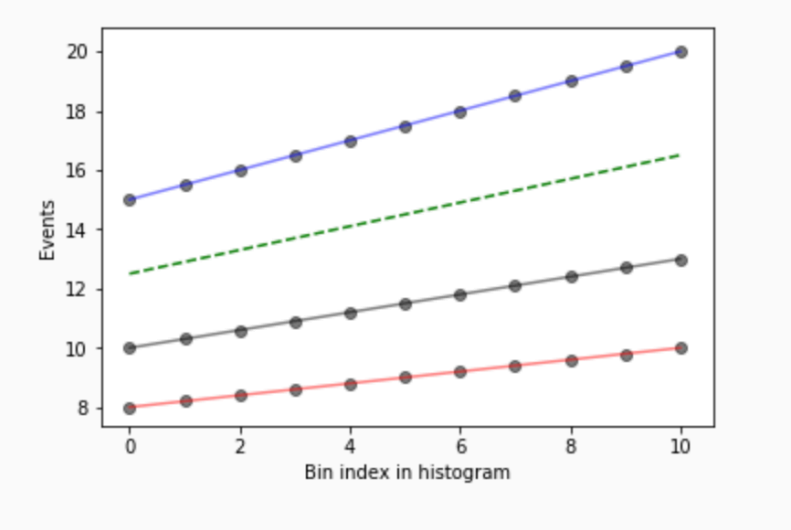
Example Systematics in HistFactory

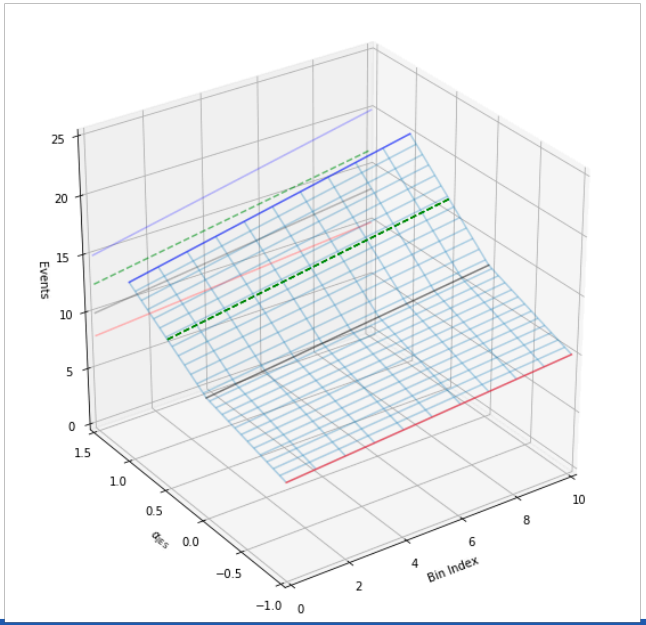
Shape uncertainties by interpolating histograms

- input: 3 histograms **f(x|-1)**, **f(x|0)**, **f(x|1)**

  down nominal up

- comput shape at arbitrary value of nuisance parameters through bin-wise interpolation **f(x|α)**
  - (Combine uses some of those interp. as well)





Good target to try to port to ML:

- **closed world, finite complexity.. it's just Gaussians, Poissons**
- **formula is more or less straight forward**

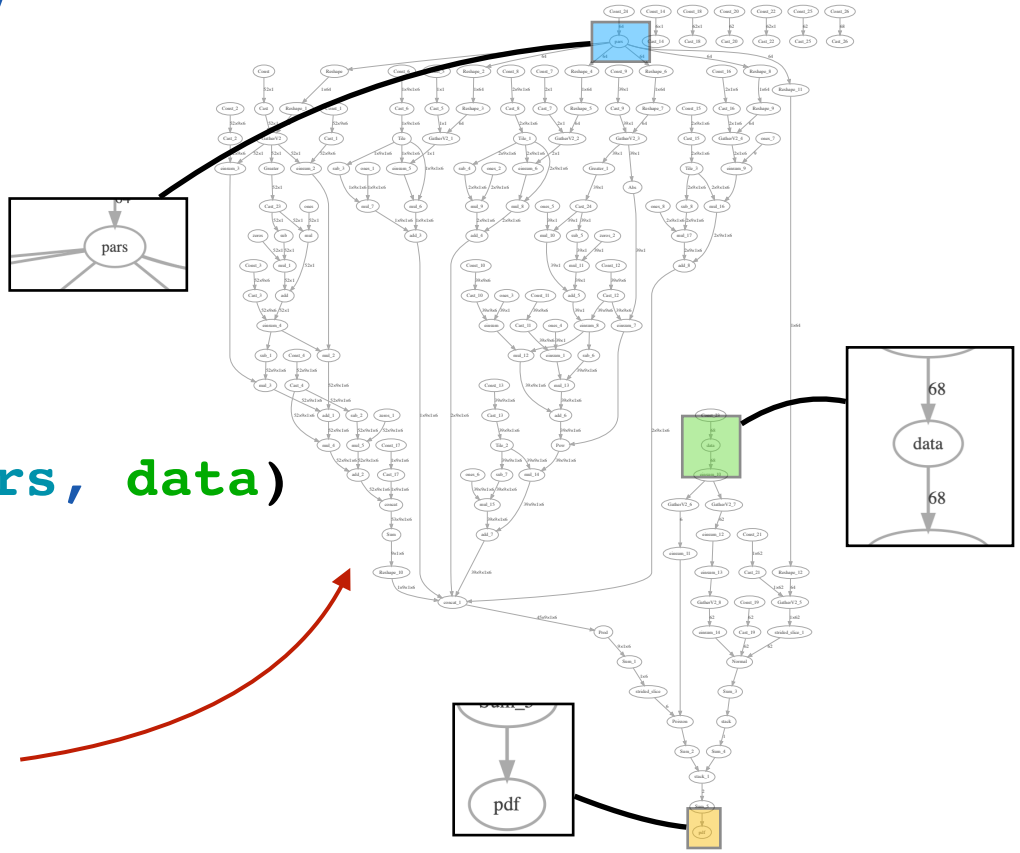
So, what does HistFacotry look like in numpy style?

Good target to try to port to ML:

- closed world, finite complexity.. it's just Gaussians, Poissons
- formula is more or less straight forward

So, what does HistFacotry look like in numpy style?

```
value = model.logpdf(pars, data)
```



Like this: O(200) fundamental ops

Transparently change to multiple ML backends

In [4]:
```
import pyhf

model = pyhf.simplemodels.hepdata_like([5],[50],[1])
pars = model.config.suggested_init()
data = [50] + model.config.auxdata
```

In [6]:
```
pyhf.set_backend(pyhf.tensor.numpy_backend())
model.logpdf(pars,data)
```

Out[6]:
```
array([-7.94210256])
```

In [7]:
```
pyhf.set_backend(pyhf.tensor.pytorch_backend())
model.logpdf(pars,data)
```

Out[7]:
```
tensor([-7.9412])
```

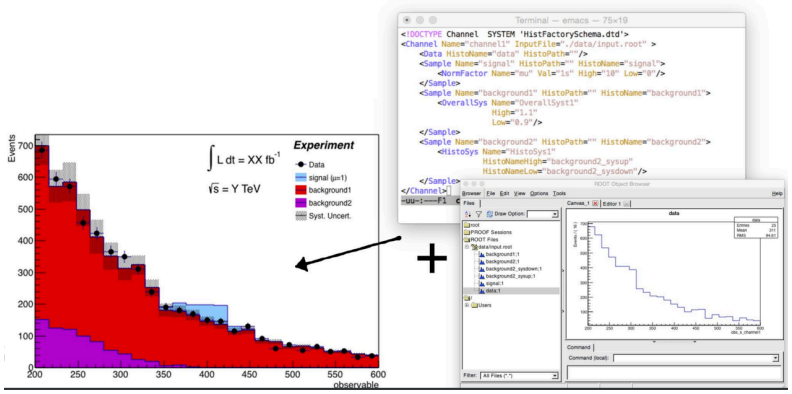
In [10]:
```
pyhf.set_backend(pyhf.tensor.tensorflow_backend())
v = model.logpdf(pars,data)

import tensorflow as tf
tf.Session().run(v)
```

Out[10]:
```
array([-7.9411774], dtype=float32)
```

In [ ]:

Providing Inputs:

HistFactory:
Historically XML files for structure, ROOT files for histograms



Combine:
text-based data cards for structure, ROOT files for histograms

20 lines (19 sloc) | 694 Bytes — Raw | Blame | Histor

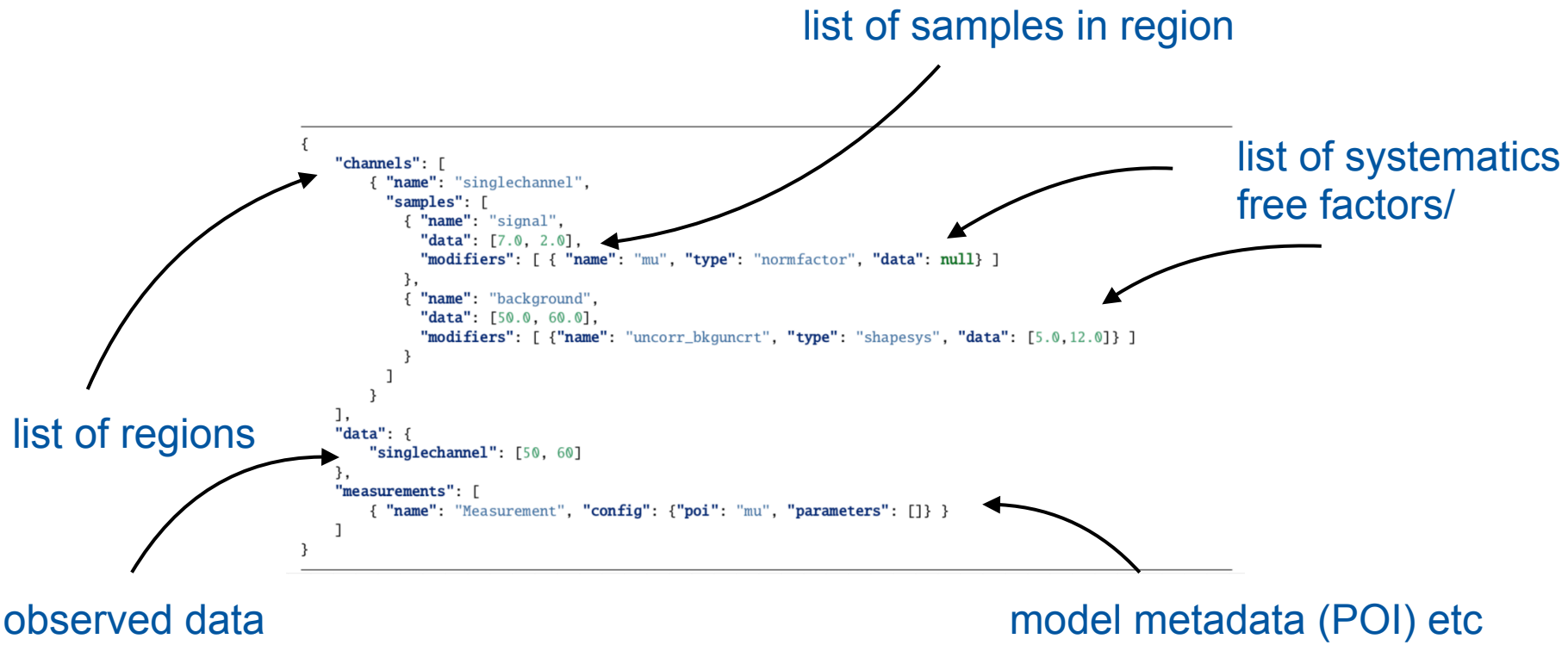
```
imax 1
jmax 1
kmax *
---------------
shapes * * simple-shapes-TH1_input.root $PROCESS $PROCESS_$SYSTEMATIC
---------------
bin bin1
observation 85
-------------------------------
bin             bin1       bin1
process         signal     background
process         0          1
rate            10         100
-------------------------------
lumi     lnN    1.10       1.0
bgnorm   lnN    1.00       1.3
alpha  shapeN2   -          1   uncertainty on background shape and normalization
sigma  shapeN2   0.5        -   uncertainty on signal resolution. Assume the histogram is a 2 sigma shift,
#                              so divide the unit gaussian by 2 before doing the interpolation
```

```
root [0]
Attaching file simple-shapes-TH1.root as _file0...
root [1] _file0->ls()
TFile**     simple-shapes-TH1.root
 TFile*     simple-shapes-TH1.root
  KEY: TH1F signal;1    Histogram of signal__x
  KEY: TH1F signal_sigmaUp;1    Histogram of signal__x
  KEY: TH1F signal_sigmaDown;1  Histogram of signal__x
  KEY: TH1F background;1    Histogram of background__x
  KEY: TH1F background_alphaUp;1    Histogram of background__x
  KEY: TH1F background_alphaDown;1  Histogram of background__x
  KEY: TH1F data_obs;1  Histogram of data_obs__x
  KEY: TH1F data_sig;1  Histogram of data_sig__x
```

When moving to ML tools, good to define a ROOT-indepdent way to define the input structure

pyhf uses JSON... inline data into the structure of the model

ROOT workspace <> JSON workspace




```
{
    "channels": [
        { "name": "singlechannel",
          "samples": [
            { "name": "signal",
              "data": [7.0, 2.0],
              "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]} ]
            }
          ]
        }
    ],
    "data": {
        "singlechannel": [50, 60]
    },
    "measurements": [
        { "name": "Measurement", "config": {"poi": "mu", "parameters": []} }
    ]
}
```

So we have:

- numpy-like implementation of HEP models
- ROOT-independent, declarative input format

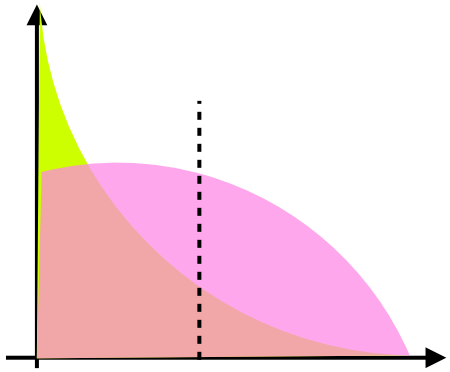
With this we can run standard HEP inference e.g. profile likelihood analysis

... using the command line

```
$> pip install pyhf
$> pyhf cls workspace.json
{
    "CLs_exp": [
        0.008897411763217407,
        0.03524468002619176,
        0.1243148689002353,
        0.3514186235832989,
        0.6941411699405086
    ],
    "CLs_obs": 0.03607409335946063
}
```

... or python APIs (note: different than using PyROOT, this does not use ROOT at all)

$$\ln \lambda(\mu) = -2\frac{L(\mu, \hat{\hat{\theta}})}{L(\hat{\mu}, \hat{\theta})}$$



$$\mathrm{CL}_s = \frac{\mathrm{CL}_{s+b}}{\mathrm{CL}_b}$$

```
[28]: import  pyhf

      model = pyhf.simplemodels.hepdata_like([5,10],[50,30],[2,3])
      data = [50,30] + model.config.auxdata
      parameters = [1.0,1.0,1.0] # nominal parameters

      model.logpdf(parameters,data) # evaluate log likelihood

[28]: array([-14.46326506])
```

```
[48]: cls = pyhf.utils.hypotest(1.0,data,model)
      cls

[48]: array([0.22800433])
```

Both Combine and HistFactory are "closed world" statistics frameworks:

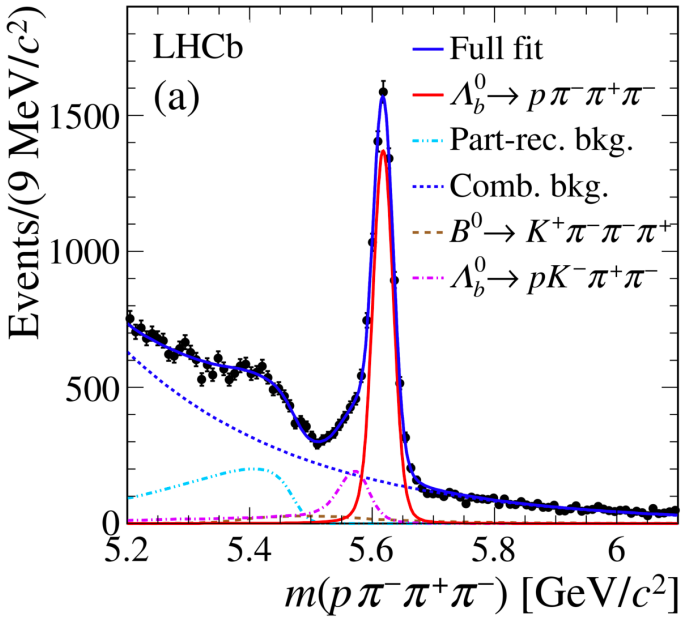
- **fixed number of building blocks, systematic types, interpolations, etc...**

This "closed-world" approach

Advantages: enables the declarative structure
Disadvantage: not straight forward to add new building blocks

RooFit: more open world

- can add custom PDFs, truncate PDFs, convolve pdfs, etc..
- used heavily for unbinned analysis (e.g. lot of LHCb stats)



the zfit project is pursuing this type of open-world appraoch using ML tools

Free-form statistical modelling in zfit

Advantage: more flexibility in modeling
Disadvantage: harder to find framework-independent representation

Example:

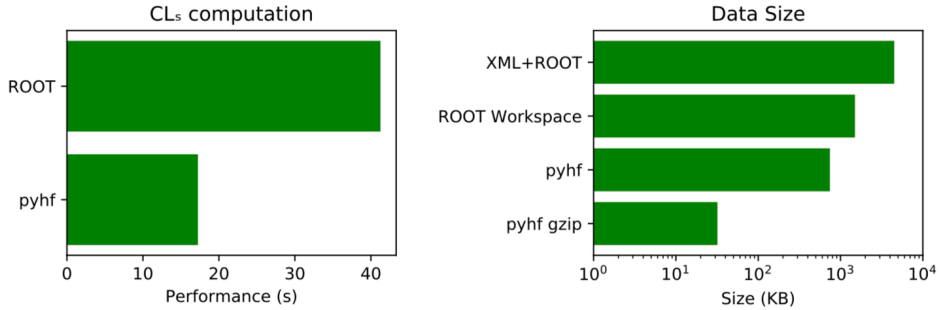
22 lines (16 sloc) | 561 Bytes

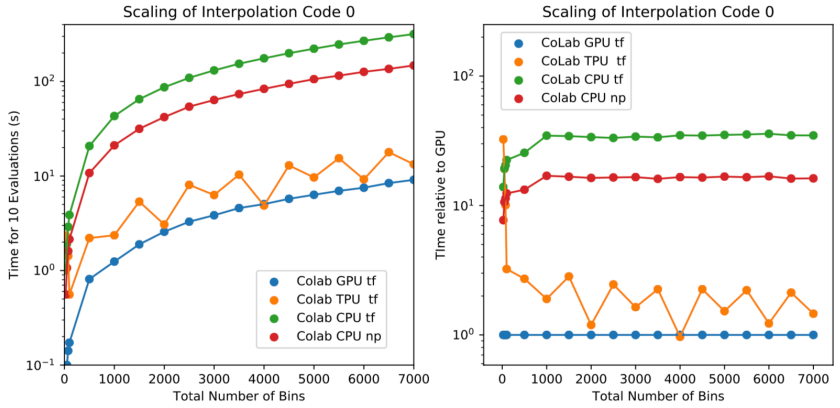
```python
#  Copyright (c) 2019 zfit

import zfit

# create space
obs = zfit.Space("x", limits=(-10, 10))

# parameters
mu = zfit.Parameter("mu", 1., -4, 6)
sigma = zfit.Parameter("sigma", 1., 0.1, 10)
lambd = zfit.Parameter("lambda", -1., -5., 0)
frac = zfit.Parameter("fraction", 0.5, 0., 1.)

# pdf creation
gauss = zfit.pdf.Gauss(mu=mu, sigma=sigma, obs=obs)
exponential = zfit.pdf.Exponential(lambd, obs=obs)

# two equivalent ways to create a sum with a fraction
sum_pdf = frac * gauss + exponential
# OR
sum_pdf = zfit.pdf.SumPDF([gauss, exponential], fracs=frac)
```

... but isn't python slower than C++ ?

HEP stats on GPUs/TPUs!

Nope, it's faster.





the trick is using the numpy-like API

- numpy ops themselves is implemented in C... fast!
- in ML libraries tensorflow, pytorch, ops are C++

Usage of hardware acceleration is automatic.

Same for Combine

Some Performance Tests

- Using current W helicity cards (1444 bins, 96 POI's, 70 nuisance parameters)

	Likelihood	Likelihood+Gradient	Hessian
Combine, TR1950X 1 Thread	10ms	830ms	-
TF, TR1950X 1 Thread	70ms	430ms	165s
TF, TR1950X 32 Thread	20ms	71ms	32s
TF, 2x Xeon Silver 4110 32 Thread	17ms	54ms	24s
TF, GTX1080	7ms	13ms	10s
TF, V100	4ms	7ms	8s

- Single-threaded CPU calculation of likelihood is 7x **slower** in Tensorflow than in Roofit (to be understood and further optimized)
- Gradient calculation in combine/Minuit is with 2n likelihood evaluations for finite differences (optimized with caching)
- Xeons are lower clocked than Threadripper, but have more memory channels and AVX-512
- Back-propagation calculation of gradients in Tensorflow is much more efficient (in addition to being more accurate and stable)

Josh Bendavid (CERN) TensorFlow Fits 12

Josh Bendavid: (talk)

Same for zfit

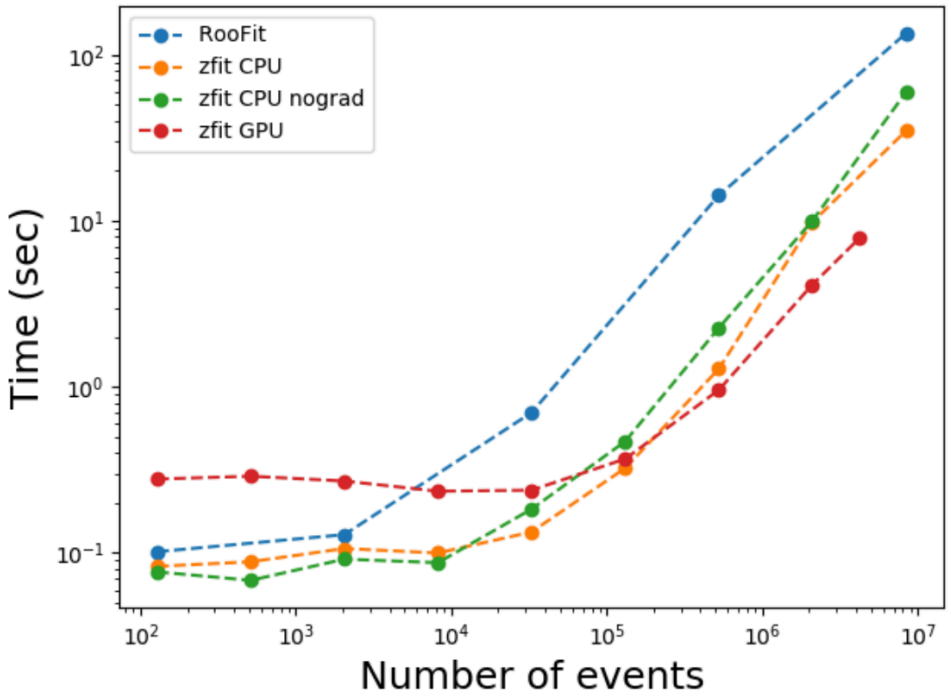
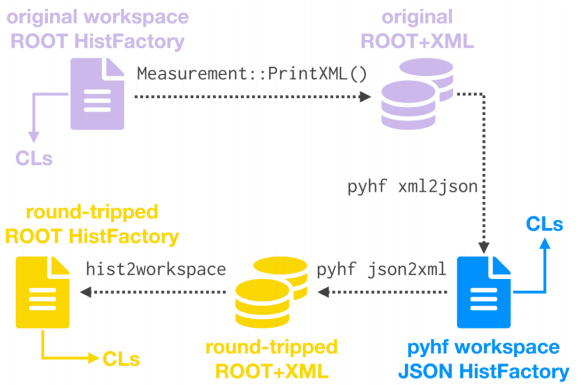


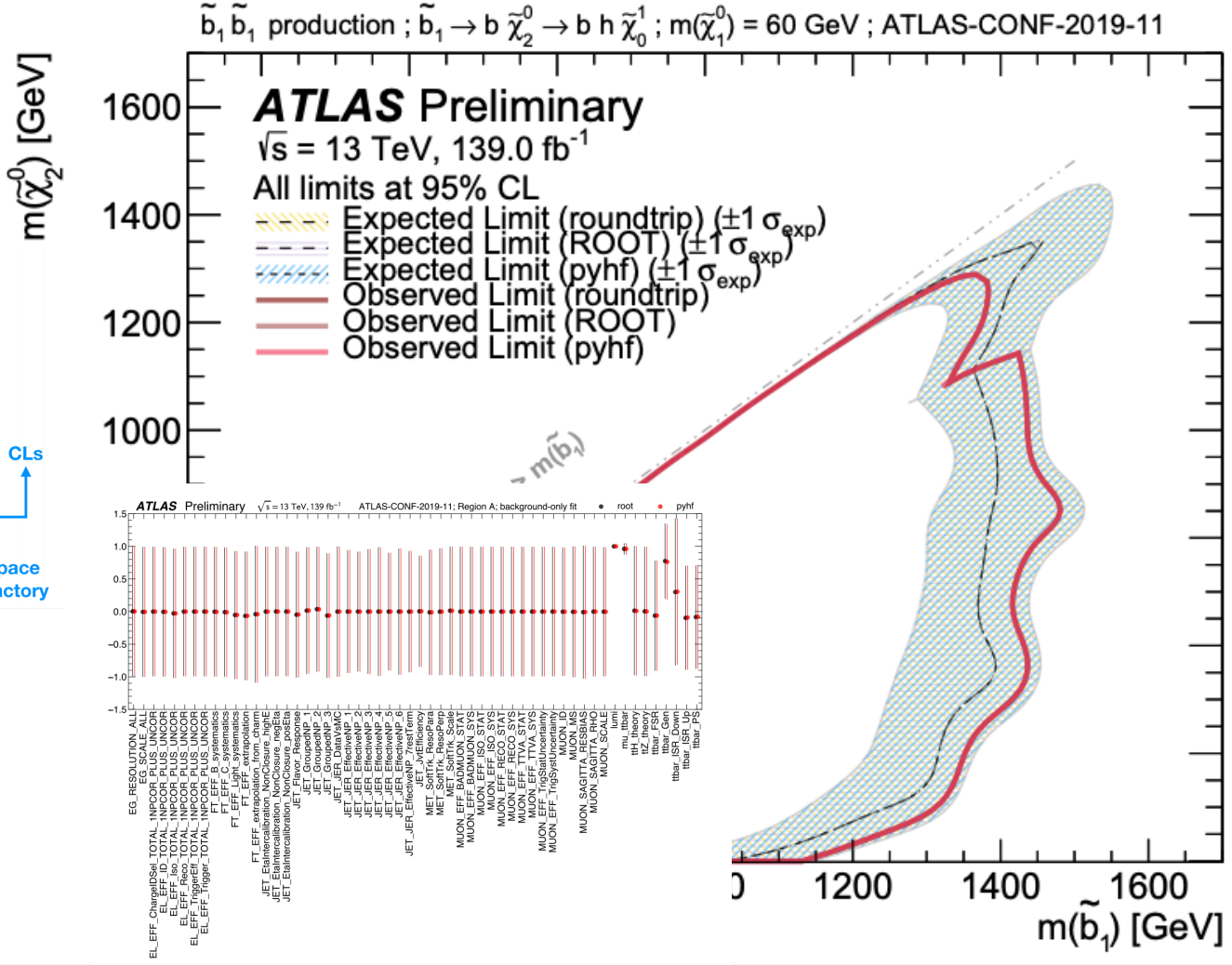
Figure A.2: Averaged time per fit for a sum of Gaussian with shared mean and width a function of the the number of generated events.

arxiv:1910.13429

How trustworthy is it? Does it reproduce results from ROOT?

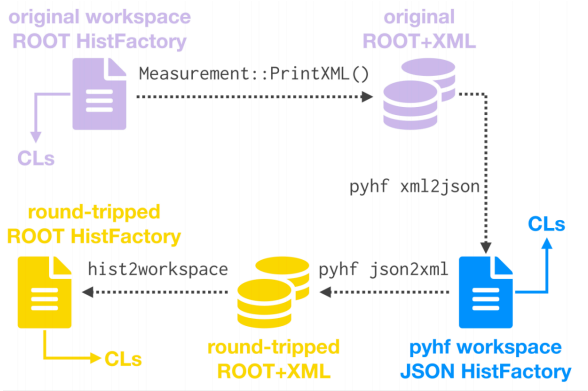
Yes.

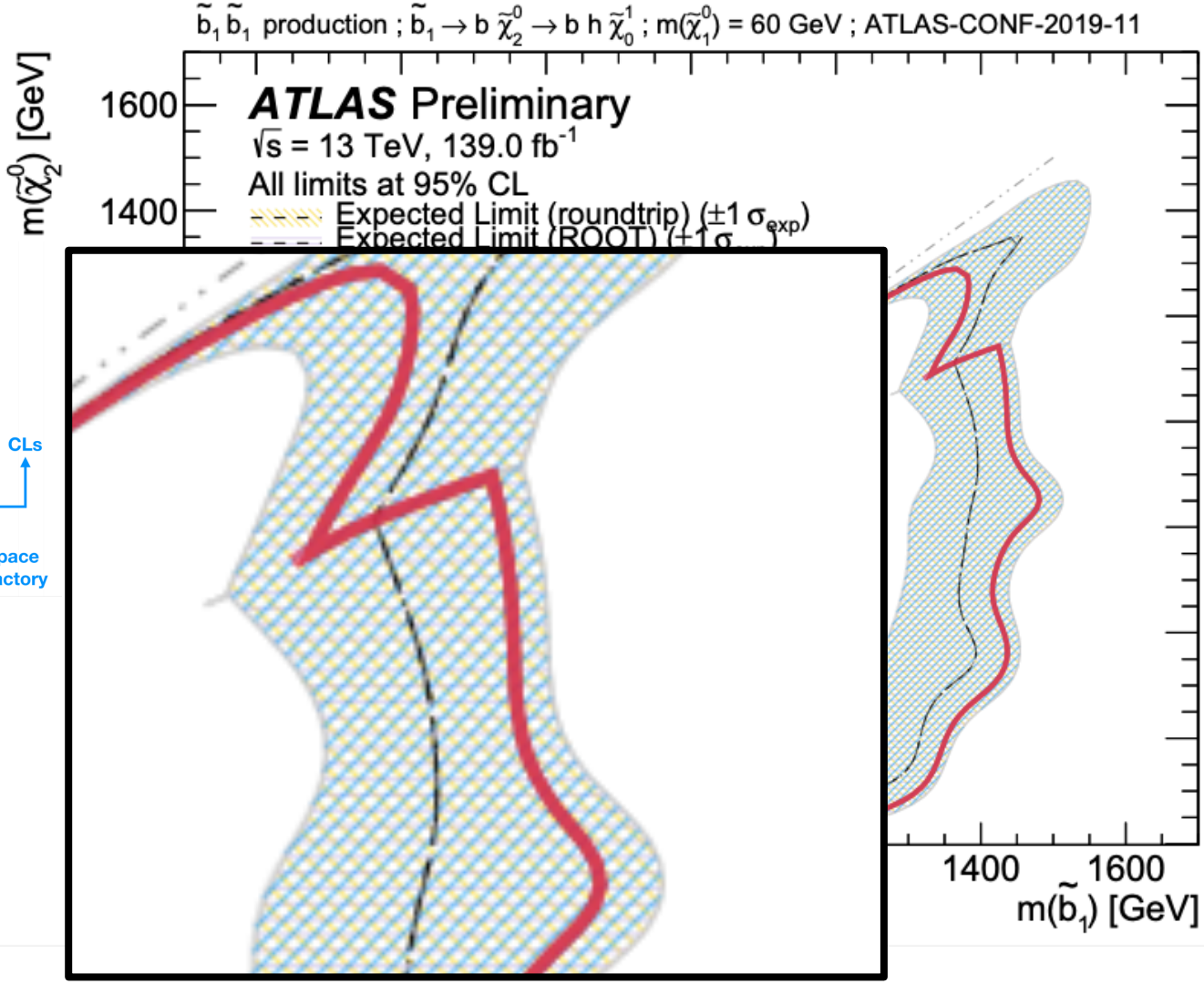




How trustworthy is it? Does it reproduce results from ROOT?

Yes.





Upshot:

re-implementing statistical models in ML tools unlocks a number of itneresting features

- **auto differentiation**
- **hardware acceleration**
- **ROOT-independent implementation**

Side-Benefit: finding representations of statistical models that are implementation-independnt (HistFactory JSON)

what can we do with this?

A small tangent

Back in Y2K:

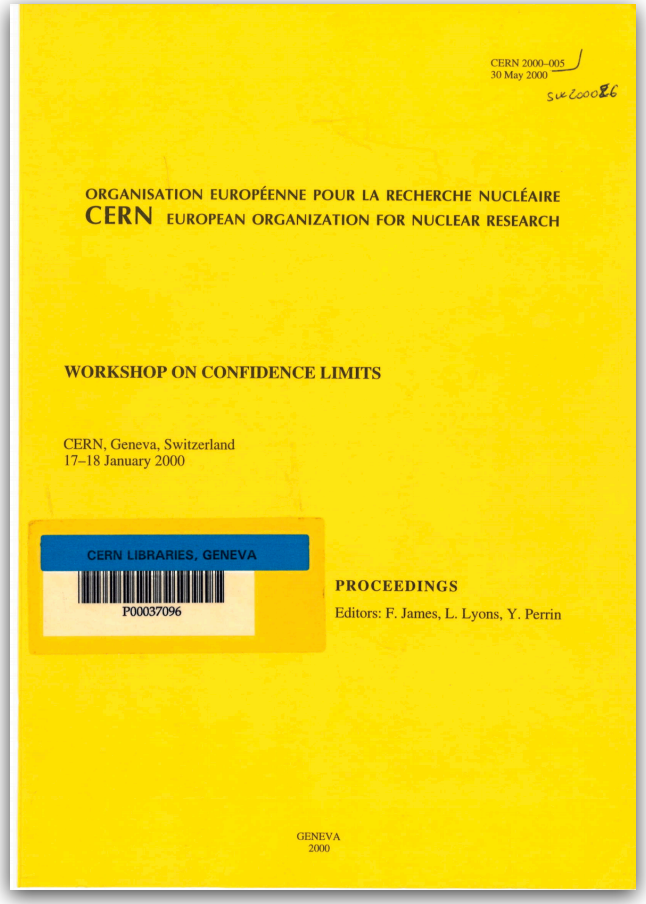
Agreement over the importance of public likelihoods:

Massimo Corradi

It seems to me that there is a general consensus that what is really meaningful for an experiment is *likelihood*, and almost everybody would agree on the prescription that experiments should give their likelihood function for these kinds of results. Does everybody agree on this statement, to publish likelihoods?

Louis Lyons

Any disagreement ? Carried unanimously. That's actually quite an achievement for this Workshop.



https://cds.cern.ch/record/452080

Information | Discussion (0) | Files

Article

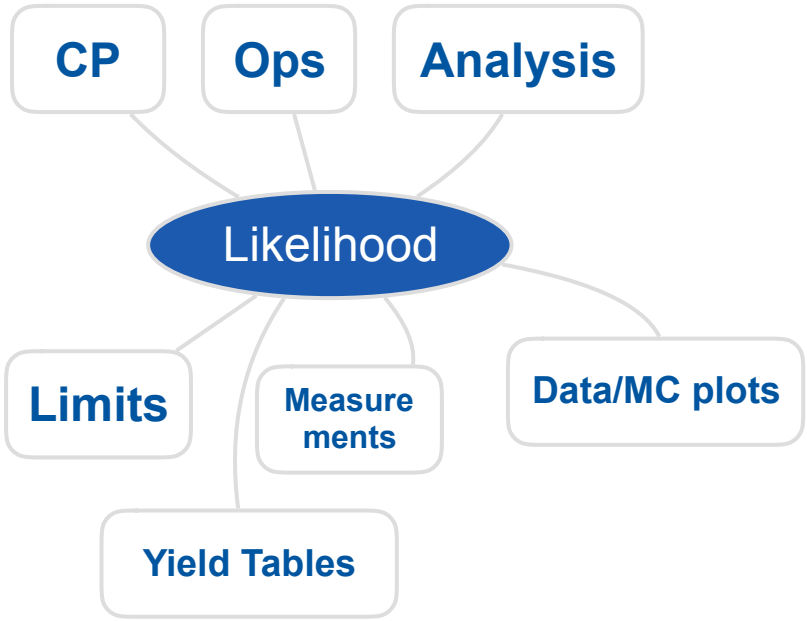
Report number	OPEN-2000-215
Title	**Panel discussion [at the 1st Workshop on Confidence Limits]**
Author(s)	Cowan, G D ; Feldman, G ; Groom, D E ; Junk, T R ; Lyons, L ; Prosper, H B
In:	1st Workshop on Confidence Limits, CERN, Geneva, Switzerland, 17 - 18 Jar
DOI	10.5170/CERN-2000-005.271
Subject category	Detectors and Experimental Techniques

Why is the likelihood important ?

- high information-density summary of analysis
  - almost no result w/o passing through it
  - a unique data fragment to preserve

$$p(\text{theory}|\text{data}) = \frac{p(\text{data}|\text{theory})}{p(\text{data})} p(\text{theory})$$

computed from the likelihood
p(data|theory)



Previously: HEP published simplified likelihoods

Information | References | Citations | Files | Plots | Data

Measurements of Higgs boson production and couplings in diboson final states with the ATLAS detector at the LHC - ATLAS Collaboration (Aad, Georges *et al.*) Phys.Lett. B726 (2013) 88-119, Erratum: Phys.Lett. B734 (2014) 406-406 arXiv:1307.1427 [hep-ex] CERN-PH-EP-2013-103

THIS DATA COMES FROM DURHAM HEPDATA PROJECT

SUMMARY:

CERN-LHC. Measurements of the cross-section times branching ratio for a standard model-like Higgs boson. The results are based on the complete pp collision data sample recorded by the ATLAS experiment at the CERN Large Hadron Collider at centre-of-mass energies of 7 TeV and 8 TeV, corresponding to an integrated luminosity of about 25 fb-1 The following table gives links to the -2ln(likelihood) values for the three channels in the (mu_ggF+ttH*B/BSM,mu_VBF+VH*B/BSM) plane for a Higgs boson mass mH = 125.5 GeV. The display link shows the data as a 2-D grid and the files are the originals from the ATLAS collaboration.

DATASETS:

Description: -2 log Likelihood for the H → γγ channel in the (μ_ggF+ttH * B/BSM, μ_VBF+VH * B/BSM) plane for a Higgs boson mass mH = 125.5 GeV.

Go to the record

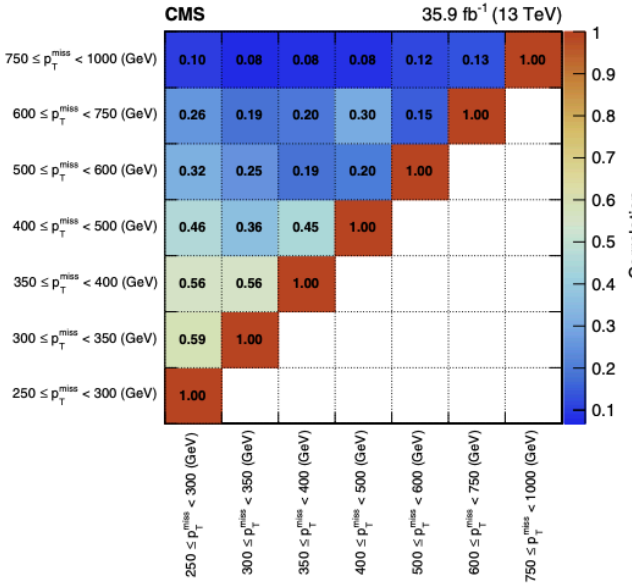
Description: -2 log Likelihood for the H→ ZZ*→ 4l channel in the (μ_ggF+ttH * B/BSM, μ_VBF+VH * B/BSM) plane for a Higgs boson mass mH = 125.5 GeV.

Go to the record

Description: -2 log Likelihood for the H→ WW*→ lνlν channel in the (μ_ggF+ttH * B/BSM, μ_VBF+VH * B/BSM) plane for a Higgs boson mass mH = 125.5 GeV.

Go to the record

K Cranmer, S Kreiss

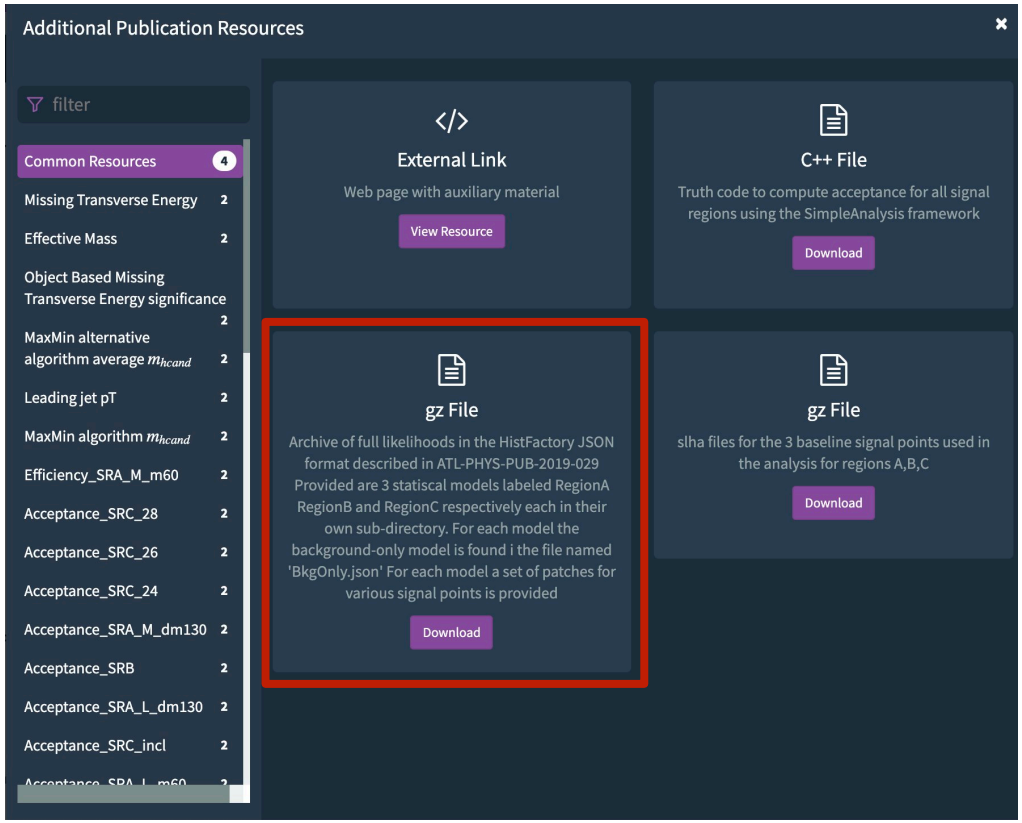


N Wardle

Reasons:

- partly sociological
- partly missing a good, software independent format to put on archives such as HepData
  - ROOT workspaces probably not?
  - now we do...

Allows anyone to reproduce a key LHC result



ATLAS EXPERIMENT

ATLAS PUB Note

ATL-PHYS-PUB-2019-029

5th August 2019

CERN

Reproducing searches for new physics with the ATLAS experiment through publication of full statistical likelihoods

The ATLAS Collaboration

Allows anyone to reproduce a key LHC result

Let's do it then

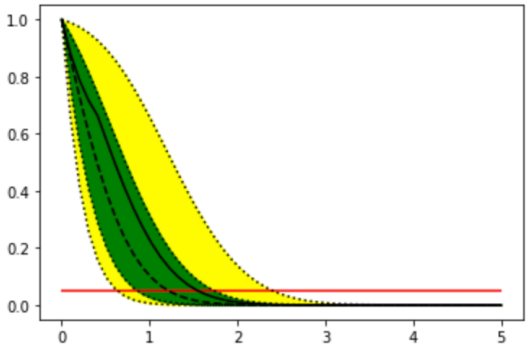
Full asymptotic profile likelihood analysis in python
Upper limits in a few lines:

[83]:
```
import numpy as np
results = []
poivals = np.linspace(0,5)
for mu in poivals:
    cls_obs, cls_exp = pyhf.utils.hypotest(mu,data,model, return_expected_set = True)
    results.append([ cls_obs[0] ] + [x[0] for x in cls_exp])

results = np.asarray(results)
print('Upper Limit (obs): μ = {:0.3}'.format(np.interp(0.05,results[:,0][::-1],poivals[::-1])))
print('Upper Limit (exp): μ = {:0.3}'.format(np.interp(0.05,results[:,3][::-1],poivals[::-1])))
```

```
Upper Limit (obs): μ = 1.55
Upper Limit (exp): μ = 1.22
```

[84]:
```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(poivals,results[:,0], c = 'black')
plt.plot(poivals,results[:,[1,-1]], linestyle = 'dotted', c = 'black')
plt.plot(poivals,results[:,[2,-2]], linestyle = 'dotted', c = 'black')
plt.plot(poivals,results[:,3], linestyle = 'dashed', c = 'black')
plt.fill_between(poivals,results[:,1],results[:,-1], color = 'yellow')
plt.fill_between(poivals,results[:,2],results[:,-2], color = 'green')
plt.hlines(0.05,0,5, color = 'r')
```

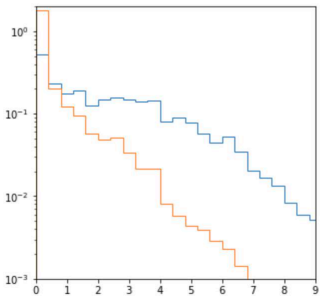
[84]:
```
<matplotlib.collections.LineCollection at 0x7f03117c1ac8>
```

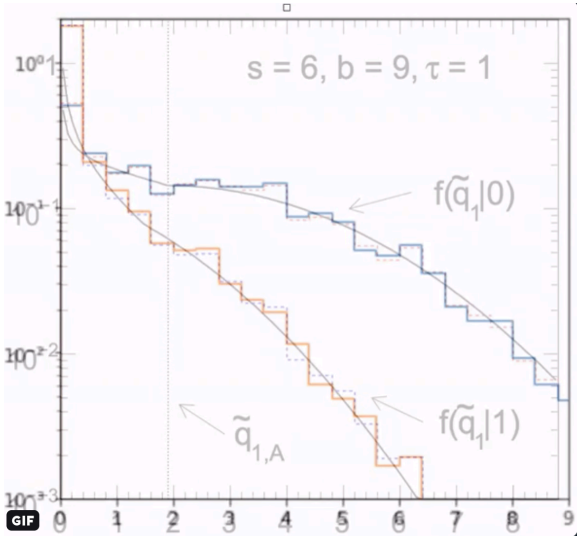


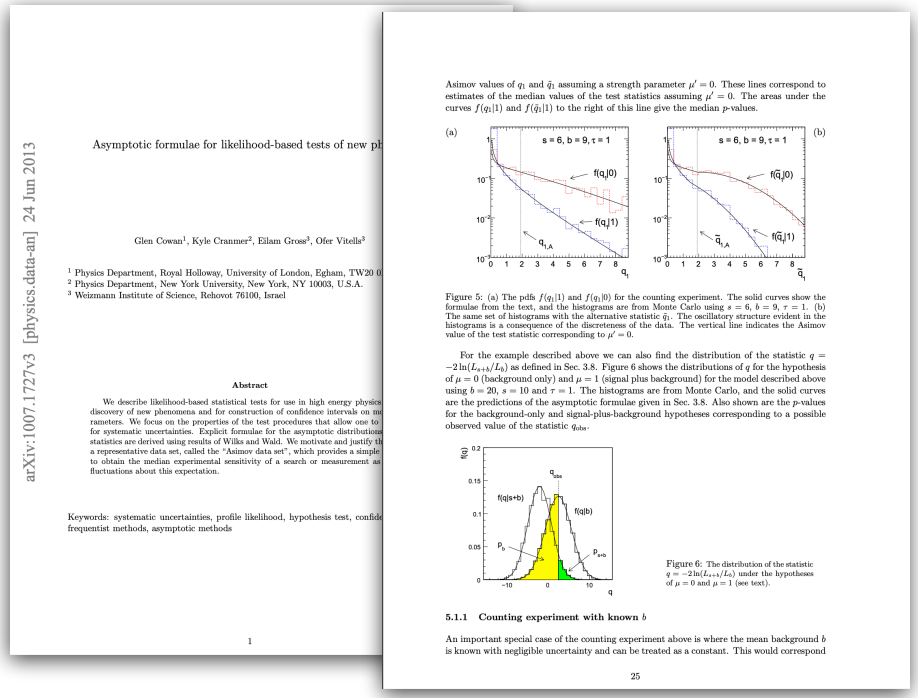
Toy-based analysis in pyhf

- **reproduce key plot in asymptotics paper**

```
In [1]: import pyhf
        import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

In [2]: m = pyhf.simplemodels.hepdata_like([6],[9],[np.sqrt(9)])

        spars = m.config.suggested_init()
        spars[m.config.poi_index] = 1.0

        bpars = m.config.suggested_init()
        bpars[m.config.poi_index] = 0.0

        spdf = m.make_pdf(spars)
        bpdf = m.make_pdf(bpars)

        bsample = bpdf.sample((20000,))
        ssample = spdf.sample((20000,))

In [3]: testb = np.asarray([pyhf.utils.qmu(1.0,s,m) for s in bsample])
        tests = np.asarray([pyhf.utils.qmu(1.0,s,m) for s in ssample])

In [4]: _,b,_ = plt.hist(np.asarray(testb), bins = np.linspace(0,10,26), histtype = 'step', density = True);
        _,b,_ = plt.hist(np.asarray(tests), bins = np.linspace(0,10,26), histtype = 'step', density = True);
        plt.xlim(0,9)
        plt.ylim(1e-3,2)
        plt.semilogy()
        plt.gcf().set_size_inches(5,5)
```







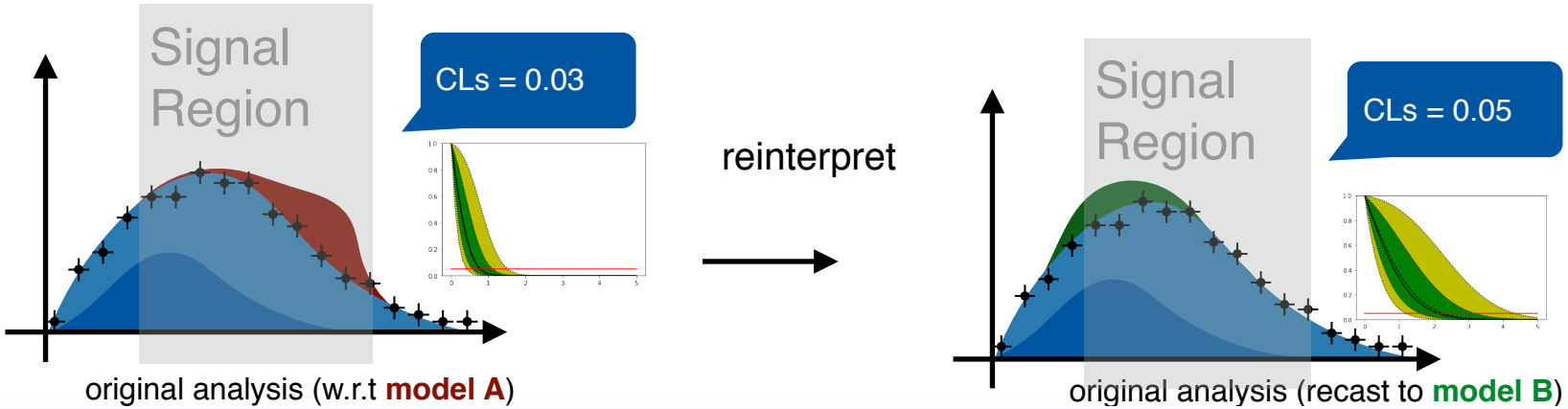
The theory argument in our likelihoods are composite hypotheses

$$p(x|\text{theory}) \rightarrow p(x|\text{theory}')$$

theory: $\text{BSM} + \text{SM}$ → theory': $\text{BSM}' + \text{SM}$

Chainging BSM → BSM' is often called "reinterpretation"

- **much cheaper than doing a dedicated analysis for that BSM'**
- **x and SM portion stay fixed (bulk of computational, human cost)**



original analysis (w.r.t **model A**)

original analysis (recast to **model B**)

The theory argument in our likelihoods are composite hypotheses

One way to think about reinterpretation:

"patching the likelihood"

$$\mathcal{L} \xrightarrow{\text{patch}} \mathcal{L}'$$

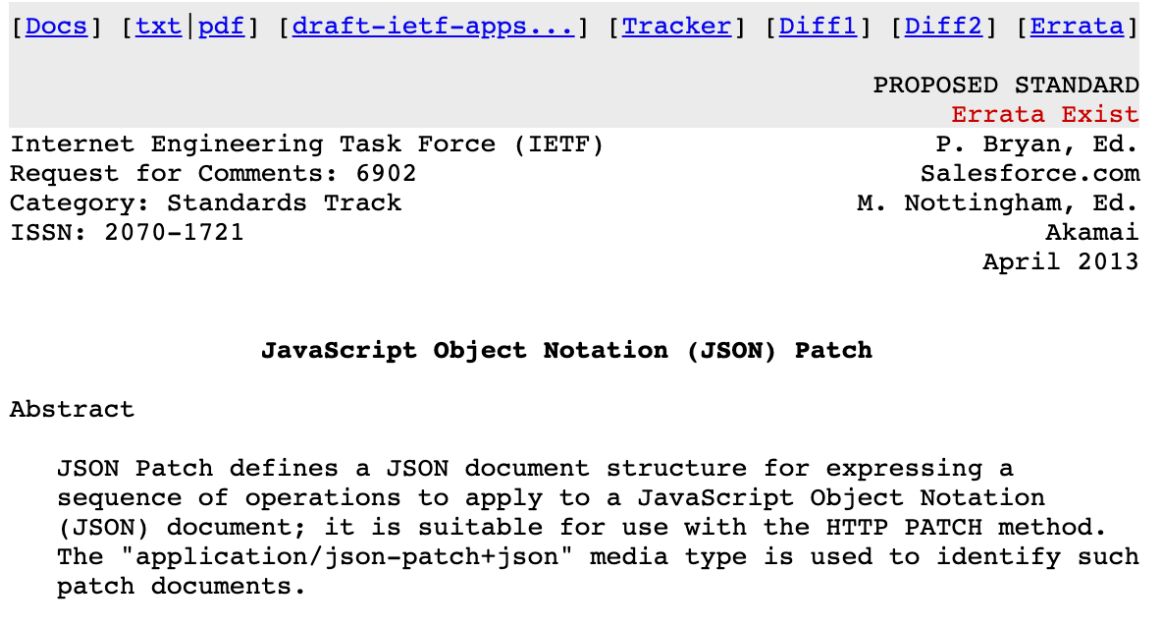
Patch operation:
- **remove old signal**
- **add new signal**

Advantage of using industry tool and declarative formats

- pre-existing technologies

patch format for JSON documents: JSONPatch

- defining a 'patch' representation for workspaces would be much harder



[Docs] [txt|pdf] [draft-ietf-apps...] [Tracker] [Diff1] [Diff2] [Errata]

PROPOSED STANDARD
Errata Exist

Internet Engineering Task Force (IETF) — P. Bryan, Ed.
Request for Comments: 6902 — Salesforce.com
Category: Standards Track — M. Nottingham, Ed.
ISSN: 2070-1721 — Akamai
April 2013

JavaScript Object Notation (JSON) Patch

Abstract

JSON Patch defines a JSON document structure for expressing a sequence of operations to apply to a JavaScript Object Notation (JSON) document; it is suitable for use with the HTTP PATCH method. The "application/json-patch+json" media type is used to identify such patch documents.

Native Reinterpretation support: test set of alternative hypotheses / RECAST by "patching the likelihood" (JSONPatch)

```
{
    "channels": [
        { "name": "singlechannel",
          "samples": [
            { "name": "signal",
              "data": [7.0, 2.0],
              "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]} ]
            }
          ]
        }
    ],
    "data": {
        "singlechannel": [50, 60]
    },
    "measurements": [
        { "name": "Measurement", "config": {"poi": "mu", "parameters": []} }
    ]
}
```

+

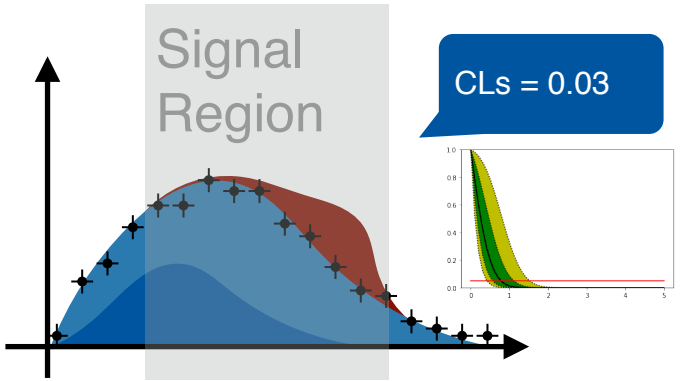
```
[{
  "op": "replace",
  "path": "/channels/0/samples/0/data",
  "value": [7.0, 2.0]
}]
```

=

```
jsonpatch likelihood.json patch.json > new.json
pyhf cls new.json
```

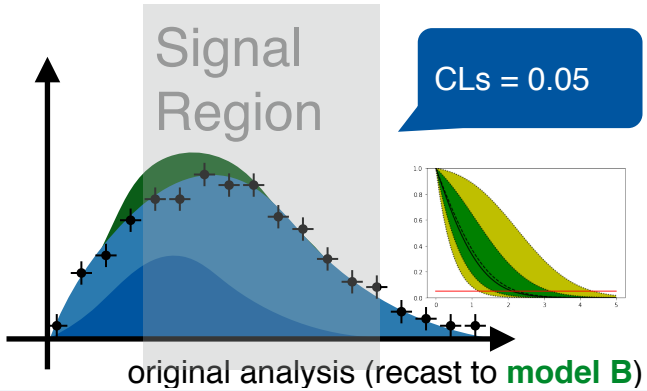
```
{
    "channels": [
        { "name": "singlechannel",
          "samples": [
            { "name": "signal",
              "data": [7.0, 2.0],
              "modifiers": [ { "name": "mu", "type": "normfactor", "data": null} ]
            },
            { "name": "background",
              "data": [50.0, 60.0],
              "modifiers": [ {"name": "uncorr_bkguncrt", "type": "shapesys", "data": [5.0,12.0]} ]
            }
          ]
        }
    ],
    "data": {
        "singlechannel": [50, 60]
    },
    "measurements": [
        { "name": "Measurement", "config": {"poi": "mu", "parameters": []} }
    ]
}
```

new yields injected



original analysis (w.r.t **model A**)

reinterpret



original analysis (recast to **model B**)

[1]:
```
import jsonpatch
import pyhf
%pylab inline
```

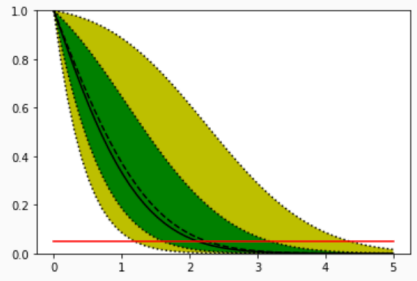
```
Populating the interactive namespace from numpy and matplotlib
```

[2]:
```
def plot_results(mutests, tests, test_size = 0.05):
    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)]
    plt.plot(mutests,cls_obs, c = 'k')
    for i,c in zip(range(5),['k','k','k','k','k']):
        plt.plot(mutests, cls_exp[i], c = c, linestyle = 'dotted' if i!=2 else 'dashed')
    plt.fill_between(testmus,cls_exp[0],cls_exp[-1], facecolor = 'y')
    plt.fill_between(testmus,cls_exp[1],cls_exp[-2], facecolor = 'g')
    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
```

The originial statistical Model

[3]:
```
data = [51, 62.]
original = pyhf.simplemodels.hepdata_like(
    signal_data =[5.,6.],
    bkg_data    =[50.,65.],
    bkg_uncerts =[5.,3.]
)
```

[4]:
```
testmus = np.linspace(0,5)
results = [
    pyhf.utils.hypotest(mu, data + original.config.auxdata,
                        original, original.config.suggested_init(), original.config.sugge
                       return_expected_set=True)
    for mu in testmus
]
plot_results(testmus,results,test_size = 0.05)
```



Patching the likelihood to replace the BSM components

A nice thing about being able to specify the entire statistical model using the ubiquitous JSON format is that we can leverage a wide ecosystem of tools to manipulate JSON documents.

In particular we can use the JSON-Patch format (a proposed IETF standard) to replace the signal component of the statistical model with a new signal.

This new signal distribution could for example be the result of a third-party analysis implementation such as Rivet.

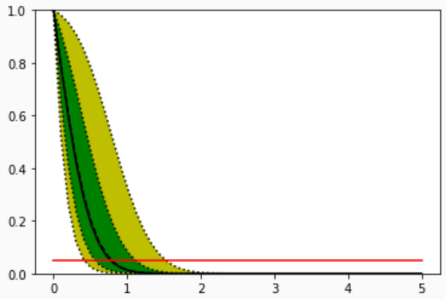
[5]:
```
new_signal = [20.,10.]
patch = jsonpatch.JsonPatch([
    {'op': 'replace', 'path': '/channels/0/samples/0/data', 'value': new_signal},
])
```

[6]:
```
recast = pyhf.Model(patch.apply(original.spec))
recast.spec
```

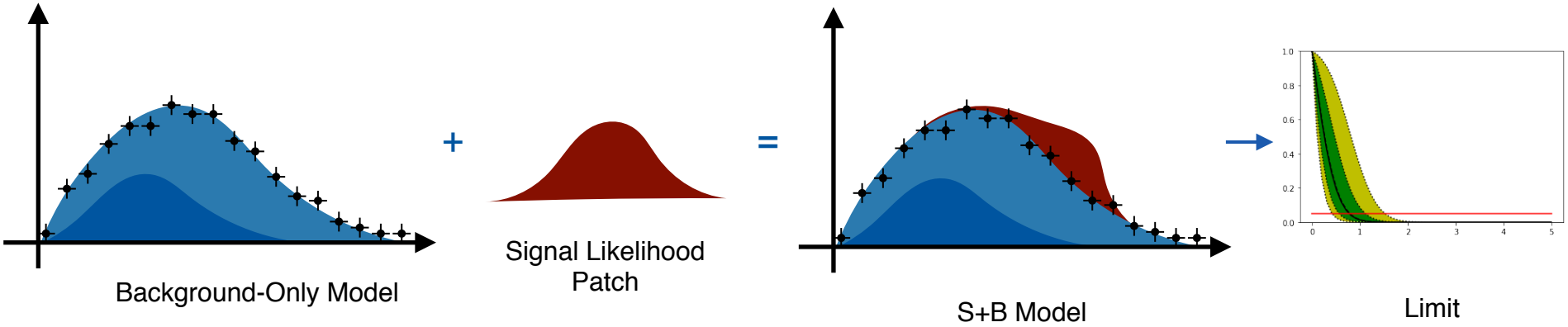
[6]:
```
{'channels': [{'name': 'singlechannel',
   'samples': [{'name': 'signal',
     'data': [20.0, 10.0],
     'modifiers': [{'name': 'mu', 'type': 'normfactor', 'data': None}]},
    {'name': 'background',
     'data': [50.0, 65.0],
     'modifiers': [{'name': 'uncorr_bkguncrt',
       'type': 'shapesys',
       'data': [5.0, 3.0]}]}]}]}
```

The Recasted Result

[7]:
```
testmus = np.linspace(0,5)
results = [
    pyhf.utils.hypotest(mu, data + recast.config.auxdata,
                        recast, recast.config.suggested_init(), recast.config.suggested_boun
                       return_expected_set=True)
    for mu in testmus
]
plot_results(testmus,results,test_size = 0.05)
```



These patches are what we use when publishing likelihoods




```
├── README.md
├── RegionA
|   ├── BkgOnly.json
|   ├── patch.sbottom_1000_131_1.json
|   ├── patch.sbottom_1000_205_60.json
|   ├── patch.sbottom_1000_230_100.json
|   ├── patch.sbottom_1000_250_60.json
|   ├── patch.sbottom_1000_330_200.json
|   ├── patch.sbottom_1000_350_60.json
|   ├── patch.sbottom_1000_430_300.json
```

```
# One signal model
$ curl -sL https://www.hepdata.net/record/resource/997020?view=true | \
  tar -O -xzv RegionA/BkgOnly.json | \
  pyhf cls --patch <(curl -sL https://www.hepdata.net/record/resource/997020?view=true | \
      tar -O -xzv RegionA/patch.sbottom_1300_205_60.json) | \
  jq .CLs_obs
0.24443635754482018

# A different signal model
$ curl -sL https://www.hepdata.net/record/resource/997020?view=true | \
  tar -O -xzv RegionA/BkgOnly.json | \
  pyhf cls --patch <(curl -sL https://www.hepdata.net/record/resource/997020?view=true | \
      tar -O -xzv RegionA/patch.sbottom_1300_230_100.json) | \
  jq .CLs_obs
0.040766025813435774
```

Conclusion:

A new breed of statistics tools based on ML tools is emerging.

- **better minimization**
- **faster**
- **integration with wider data science world**
- **driving open data advancements**

It's not as polished yet as RooFit, but a gret time to get involved, shape the future of HEP statistics tools