Introduction to cabinetry

Alexander Held¹

¹ University of Wisconsin-Madison

Belle II pyhf workshop
https://indico.belle2.org/event/8470/
March 3, 2023

This work was supported by the U.S. National Science Foundation (NSF) Cooperative Agreement OAC-1836650 (IRIS-HEP).
Intro: constructing and using statistical models

- **Disclaimer:** I am working with ATLAS and probably have a biased view! Curious to learn where things differ in Belle II.

- **Binned template fits** are widely used for **statistical inference**

- **Statistical models** used in particle physics are often **rather complex**
  - lots of book-keeping to handle $O(10k)$ histograms for typical ATLAS applications
  - frequent model modifications needed for tests & debugging

- A set of **tools** emerged over time to aid with **model construction** and **inference**
  - In ATLAS: [HistFitter](https://www.hepforge.org/tools/histfitter/) and many more internal tools, [Combine](https://twiki.cern.ch/twiki/bin/view/CMSPublic/CombineH矣#Overview) for CMS
    - interested to learn more about what is used elsewhere
  - (some of) these tools also provide utilities to visualize inference result & simplify debugging
The cabinetry library

- **cabinetry** is a modern Python library for constructing and/or operating **HistFactory** models
  
  ```
pip install cabinetry
  ```
  
  - uses **pyhf**, integrates seamlessly with the Python HEP ecosystem
  - modular design: use the pieces of **cabinetry** you need
  - part of the [Scikit-HEP](https://github.com/scikit-hep) project

- **cabinetry ↔ pyhf** is roughly like **HistFitter ↔ ROOT (RooFit, HistFactory, RooStats)**
Working with cabinetry

- **cabinetry** is a **Python library** for creating and operating HistFactory models
  - design and **construct statistical models** (workspaces) from instructions in **declarative configuration**
    - analyzers specify selections for signal/control regions, (Monte Carlo) samples, systematic uncertainties
    - **cabinetry** steers creation or collects provided **template histograms** (region $\otimes$ sample $\otimes$ systematic)
    - **cabinetry** produces **HistFactory workspaces** (serialized fit model)
  - perform **statistical inference**
    - including diagnostics and visualization tools to study and disseminate results
Designing a statistical model

- **Declarative configuration** (JSON/YAML/dictionary) specifies everything needed to build a workspace
  - can concisely capture complex `region ⊗ sample ⊗ systematic` structure

```
General:
  Measurement: "Example"
  InputPath: "input/{SamplePaths}"  
  HistogramFolder: "histograms/"
  POI: "Signal_norm"

Regions:
  - Name: "Signal_region"
    Filter: "nJets >= 8"
    Variable: "jet_pt"
    Binning: [200, 300, 400, 500]

Samples:
  - Name: "Data"
    SamplePaths: "data.root"
    Tree: "events"
    Data: True
  - Name: "Signal"
    SamplePaths: "signal.root"
    Tree: "events"
    Weight: "weight_nominal"
  - Name: "Background"
    SamplePaths: "background.root"
    Tree: "events"
    Weight: "weight_nominal"

Systematics:
  - Name: "Luminosity"
    Up:
      Normalization: 0.05
    Down:
      Normalization: -0.05
    Samples: ["Signal", "Background"]
    Type: "Normalization"

- Name: "ModelingVariation"
  Up:
    Tree: "events_up"
    Weight: "weight_modeling"
  Down:
    Tree: "events_down"
    Weight: "weight_modeling"
    Smoothing:
      Algorithm: "3SOM, twice"
      Samples: "Background"
      Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```
Template histograms and workspace building

- **Workspaces construction** happens in three steps:
  1) create template histograms from columnar data following config instructions
     - backends execute instructions (default: uproot, experimental: coffea)
     - alternatively: collect existing user-provided histograms
  2) optional: apply post-processing to templates (e.g. smoothing)
  3) assemble templates into workspace (JSON file)

- Utilities provided to visualize and debug fit model

---

**event yield table**

<table>
<thead>
<tr>
<th>sample</th>
<th>Control region</th>
<th>Signal region</th>
</tr>
</thead>
<tbody>
<tr>
<td>single top</td>
<td>44.74</td>
<td>8.35</td>
</tr>
<tr>
<td>ttbar</td>
<td>635.98</td>
<td>13.28</td>
</tr>
<tr>
<td>z_tth</td>
<td>38.90</td>
<td>1.00</td>
</tr>
<tr>
<td>total</td>
<td>711.61 ± 28.28</td>
<td>15.43 ± 2.69</td>
</tr>
<tr>
<td>data</td>
<td>713.08</td>
<td>14.00</td>
</tr>
</tbody>
</table>

**fit model visualization**

**Visualization of individual template histograms**

**fit to data**
Implementations for common inference tasks exist

- includes associated visualizations

**likelihood scans**

**discovery significance**

**parameter correlations**

**nuisance parameter pulls**

**upper parameter limits**

**nuisance parameter impacts**

Alexander Held
Working with an unknown workspace

• Pick a **workspace** from [HEPData: 10.17182/hepdata.89408.v3](10.17182/hepdata.89408.v3) (analysis: JHEP 12 (2019) 060)

  - download workspace with `pyhf`
  - perform inference and visualize results with `cabinetry`

  • can use inference features regardless of how a workspace was built, **functionality factorizes**!

• See arXiv:2109.04981 and try it on Binder
cabinetry: summary

- **cabinetry** is
  - a modular Python library to create and/or operate statistical models for inference with template fits
  - built upon the powerful and growing Python HEP ecosystem
  - using a slightly different design approach to other tools: more library, less framework
    - analyzers will generally need to write some code: hopefully less “black box” and more flexible, but more work
Backup
The HistFactory model
The HistFactory model: overview

• **HistFactory** is a statistical model for **binned template fits**
  • prescription for constructing probability density functions (pdfs) from small set of building blocks
  • covers wide range of use cases
  • models can be serialized to *workspaces*

\[
p(\vec{n}, \vec{a} | \vec{k}, \vec{\theta}) = \prod_{i} \text{Pois}(n_i | \nu_i(\vec{k}, \vec{\theta})) \cdot \prod_{j} c_j(a_j | \theta_j)
\]
The HistFactory model specifies how to construct the **likelihood function** from a set of building blocks:

- **Channels** (also called regions sometimes) are regions of phase space
- Distributions of **samples** (MC and data) in channels are provided by template histograms
- **Systematics** act on samples and are specified via the distribution at ±1σ shifts

\[
p(\vec{n}, \vec{a} \mid \vec{k}, \vec{\theta}) = \prod_i \text{Pois}(n_i \mid \nu_i(\vec{k}, \vec{\theta})) \cdot \prod_j c_j(a_j \mid \theta_j)
\]
Systematic uncertainties with HistFactory

- **Common systematic uncertainties** specified with two template histograms
  - "up variation": model prediction for $\theta = +1$
  - "down variation": model prediction for $\theta = -1$
  - interpolation & extrapolation provides model predictions $\nu$ for any $\bar{\theta}$

- **Gaussian constraint terms** used to model auxiliary measurements (in most cases)
  - centered around nuisance parameter (NP) $\theta_j$
  - normalized width ($\sigma = 1$) and mean (auxiliary data $a_j = 0$)
  - penalty for pulling NP away from best-fit auxiliary measurement value

$$p(\vec{n}, \vec{a} | \vec{k}, \vec{\theta}) = \prod_i \text{Pois}(n_i | \nu_i(\vec{k}, \vec{\theta})) \cdot \prod_j c_j(a_j | \theta_j)$$
The HistFactory model: structure

• **HistFactory** models follow a specific structure
  • a list of phase space regions: *channels* (defined by event selection, can have one or multiple bins)
  • each *channel* contains a list of *samples* (different type of physics processes)
  • each *sample* is affected by a list of *modifiers* (e.g. parameters of interest (POIs) or encoding systematic uncertainties)
    - *modifiers* with the same name are controlled by the same parameter and thus correlated
  • plus measurement configuration (e.g. “hold this parameter constant” and observations (e.g. real data))
Normalizing histosys modifiers

• Due to the use of linear extrapolation, histosys modifiers can cause negative yield predictions

  • example: Gist

  • (partial) solution: split overall channel normalization effect into correlated normsys [OverallSys]

```
"data": [10, 5],
"modifiers": [
  {
    "data": {"hi_data": [14, 9], "lo_data": [6, 1]},
    "name": "Histosys_example",
    "type": "Histosys",
  },
],

pure histosys

exact match where templates are defined (green points) by design

 correlated histosys + normsys
```

Alexander Held
More about pyhf
HistFactory: workspace formats

- Until 2018, the HistFactory model had only been implemented in ROOT
  - using RooFit, with RooStats available for statistical inference
  - workspaces can be serialized as
    - xml (model structure) + ROOT files (histograms) or single ROOT file (generic RooFit workspace)
    - experimental: JSON* (see Carsten Burgard’s ROOT Users workshop contribution)

- pyhf introduced a new format: workspaces serialized to JSON
  - JSON format used by ATLAS to publish models on HEPData (list of public models)
    - JSON Patch to swap out workspace components (e.g. signal model)
    - versioned JSON schema describes the declarative model

*: more generic than the pyhf format, not HistFactory-specific
A HistFactory JSON workspace with pyhf

- JSON structure maps directly to workspace structure
  - highly human-readable!

```json
{  "channels": [
    {  "name": "SR",  
      "samples": [
        {  
          "data": [10.0, 15.0],  
          "modifiers": [
            {  "data": null,  "name": "mu",  "type": "normfactor"}
          ],  
          "name": "Signal"
        },
        {  "data": [50.0, 45.0],  
          "modifiers": [
            {  "data": {  "hi": 1.1,  "lo": 0.9},  "name": "Modeling_unc",  "type": "nomsys"}
          ],  
          "name": "Background"
        }
      ],  
      "measurements": [
        {  "config": {  "parameters": [],  "pol": "mu"},  
           "name": "minimal_example"
        }
      ],  
      "observations": [{  "data": [68.0, 68.0],  "name": "SR"},  
                       "version": "1.0.0"
      ]
    }
  ]
}
```

- single channel
- two samples
- modifiers
- measurement configuration
- observed data
Model patching

• Especially in searches, it is common to use **many different models that slightly differ**
  • same background model but many different signal hypotheses (e.g. different resonance masses)

• It is possible to **edit and swap out pieces of a workspace via JSON Patch**
  • e.g. add a **new component** to your model
  • or replace your signal model

figure credit: Lukas Heinrich
HistFactory: implementations

- **Until 2018**, the HistFactory model had only been implemented in **ROOT**
  - using RooFit, with RooStats available for statistical inference

- **pyhf** implements the HistFactory model in **pure Python** (pip install pyhf)
  - leverages tensor backends: efficient vectorized calculations & hardware acceleration
    - can automatically differentiate through statistical model (computational graph)
      - exact gradients for minimizers
      - enables end-to-end analysis optimization: **neos**
    - backend-agnostic API (and CLI)

**example**: autodiff through model yield prediction (e.g. for uncertainty propagation)
*it just works!*
pyhf: where to find more

- pyhf:
  - can be installed via $ pip install pyhf
  - pyhf[backends] for all tensor backends
  - is open source and
    - developed on GitHub: scikit-hep/pyhf
    - published on PyPI
    - documented on Read the Docs
      * contains links to talks / paper as well
    - provides tutorials: pyhf.github.io/pyhf-tutorial
HistFactory: closure

- Consistent results of ROOT and pyhf implementations demonstrated with many examples

closure tests from ATL-PHYS-PUB-2019-029
pyhf: summary

- pyhf provides
  - a declarative JSON schema for workspaces, used for statistical model publication and reinterpretation
  - a HistFactory implementation in Python that leverages tensor backends

- pyhf is a library exposing an API providing relevant functionality also found in RooFit, HistFactory and RooStats
  - it does not provide high-level functionality which applications like HistFitter focus on
  - examples of things the pyhf API provides:
    - model yield prediction & NLL given parameters, details about model structure, MLE, workspace pruning
  - examples of things not in scope for pyhf:
    - post-fit model prediction plots, nuisance parameter ranking
Example: expected_data and MLE fits

- Example: model predictions and maximum likelihood estimates

```python
In [2]: import pyhf

ws = pyhf.Workspace(spec)
model = ws.model()  # the statistical model
data = ws.data(model)  # observed data

data # includes auxiliary data! 0.0 here for the Modeling_unc NP

Out[2]: [60.0, 60.0, 0.0]

In [3]: # list the available model parameters:
   model.config.par_names()

Out[3]: ['mu', 'Modeling_unc']

In [4]: # nominal model prediction
   model.expected_data(model.config.suggested_init(), include_auxdata=False)

Out[4]: array([60., 60.])

In [5]: # model prediction with the Modeling_unc parameter set to 1.0 and mu=0
   model.expected_data([0.0, 1.0], include_auxdata=False)

Out[5]: array([55. , 49.5])

Why is the first bin 55? Nominal background yield is 50 (no signal contribution since mu=0),
scaled by 1.1 (Modeling_unc = 1).

In [6]: # perform a maximum likelihood fit of the model to data
   fit_results = pyhf.infer.mle.fit(data, model)
   for parname, result in zip(model.config.par_names(), fit_results):
     print(f"{parname} = {result.join(f"{result}"")}

mu = 1.0
Modeling_unc = 0.0
```
Missing / incomplete features of interest for pyhf

- **Expression for normalization factors** [pyhf#850 + pyhf#1627]
  - to scale samples by (arbitrary) functions of normalization factors (instead of just linearly)
  - technically possible now, but requires boilerplate

- **Multi-POI support** [pyhf#179]
  - simple to work around, mostly used as metadata
  - requires schema change to support list of strings in workspace (**ROOT** uses single string, list is arguably better)

- **Configurable constraint terms** [pyhf#1829 & constraint term removal pyhf#820]
  - related bug in **ROOT** until recently for removing constrained terms with **HistoSys** [root#9070]

- **Staterror pruning** [pyhf#662 + pyhf#760]
  - cannot prune per-bin currently, pruning information not saved in **JSON**

- **Interpolation codes stored in workspace** ([pyhf#1762](https://github.com/timvandevall/pyhf/pull/1762) is related)
  - not currently stored in **ROOT** workspace with xml+root either, but arguably should be to fully specify model
Error propagation with `pyhf`

- **Code example** to give a better feeling for the `pyhf` API
  - this does both error propagation & bootstrapping
  - few lines each + boilerplate* to set everything up

- **Comparison:** [cabinetry#221](cabinetry#221)
  - choice of method can have a non-negligible impact

```python
import json
import pathlib
import numpy as np
import pyhf

# get statistical model + data
fname = pathlib.Path("example_workspace.json")
spec = json.loads(fname.read_text())
ws = pyhf.Workspace(spec)
model = ws.model()
data = ws.data(model)

# fit with pyhf
pyhf.set_backend(pyhf.tensorlib, "minuit")
result, result_obj = pyhf.infer. fit(data, model, return_result_obj=True)

# error propagation
y, ycov = jacobi.propagate
lambda p, model.expected_data(p, include_auxdata=False),
result_obj.minuit.values,
result_obj.minuit.covariance,
)
print("Error propagation:\nyield: \{y\}\nuunc: \{np.diag(ycov)** 0.5\}\n")

# bootstrap sampling
rng = np.random.default_rng()
par_b = rng.multivariate_normal
(result_obj.minuit.values, result_obj.minuit.covariance, size=50000)
y_b = [model.expected_data(p, include_auxdata=False) for p in par_b]
yerr_boot = np.std(y_b, axis=0)
print("Error bootstrapping:\nyield: \{np.mean(y_b, axis=0)\}\nuunc: \{yerr_boot\}")
```

* can skip the remaining boilerplate code with `cabinetry`
import json
import pathlib

import jacobi
import numpy as np
import pyhf

# get statistical model + data
fname = pathlib.Path("example_workspace.json")
spec = json.loads(fname.read_text())
ws = pyhf.Workspace(spec)
model = ws.model()
data = ws.data(model)

# fit with pyhf
pyhf.set_backend(pyhf.tensorlib, "minuit")
result, result_obj = pyhf.infer.mle.fit(data, model, return_result_obj=True)

# error propagation
y, ycov = jacobi.propagate(
    lambda p: model.expected_data(p, include_auxdata=False),
    result_obj.minuit.values,
    result_obj.minuit.covariance,
)
print("via error propagation:
    yield: {y}
    unc: {np.diag(ycov)** 0.5}"")

# bootstrap sampling
rng = np.random.default_rng(1)
par_b = rng.multivariate_normal(result_obj.minuit.values, result_obj.minuit.covariance, size=50000)
y_b = [model.expected_data(p, include_auxdata=False) for p in par_b]
yerr_boot = np.std(y_b, axis=0)
print("via bootstrapping:
    yield: {np.mean(y_b, axis=0)}
    unc: {yerr_boot}"
More about cabinetry
• **cabinetry:**

  - can be installed via `$ pip install cabinetry`

  - `cabinetry[contrib]` for extra features

  - is open source and

    - developed on GitHub: [scikit-hep/cabinetry](https://github.com/scikit-hep/cabinetry)

    - published on [PyPI](https://pypi.org/project/cabinetry/)

    - documented on [Read the Docs](https://cabinetry.readthedocs.io/en/latest/)

      - contains links to talks / paper as well

    - provides tutorials: [cabinetry/cabinetry-tutorials](https://cabinetry.readthedocs.io/en/latest/cabinetry-cabinetry-tutorials/)
Future directions for cabinetry

• **Next steps and goals:**
  - nuisance parameter pruning (#311)
  - performance improvements for workspace construction from ntuple inputs
  - further improvements to plotting API (#265)
  - longer term: support end-to-end automatic differentiation (#233)
    - optimize analysis selection and design via gradient descent, see neos ([PyHEP 2020 talk](#))
  - your ideas?
    - get involved: from feedback to development, your contributions are welcome!
pyhf and cabinetry within the broader ecosystem
Why cabinetry?

- **Why cabinetry?**
  - pure Python and no ROOT dependency, fills gap in Python ecosystem
  - modular approach: avoid lock-in
    - benefit from growing columnar analysis ecosystem (*coffea* etc.)
  - openly developed, fully available to broader community beyond a specific experiment
  - follow good practices with extensive automated testing (see [coverage](#))
  - chance to take different design decisions informed by years of experience with existing tools
    - decouple fit model specification and implementation
    - declarative approach, but allow custom code injection at core steps in the workflow

- **Why the name?**
  - a workspace is like a cabinet: it organizes data into many bins (like drawers in a cabinet)
  - the building of these “workspace cabinets” is *cabinetry*