Building and steering binned template fits with cabinetry

Kyle Cranmer[1] and *Alexander* Held[1,*]

[1]New York University, United States

Abstract. The `cabinetry` library provides a Python-based solution for building and steering binned template fits. It tightly integrates with the pythonic High Energy Physics ecosystem, and in particular with `pyhf` for statistical inference. `cabinetry` uses a declarative approach for building statistical models, with a JSON schema describing possible configuration choices. Model building instructions can additionally be provided via custom code, which is automatically executed when applicable at key steps of the workflow. The library implements interfaces for performing maximum likelihood fitting, upper parameter limit determination, and discovery significance calculation. `cabinetry` also provides a range of utilities to study and disseminate fit results. These include visualizations of the fit model and data, visualizations of template histograms and fit results, ranking of nuisance parameters by their impact, a goodness-of-fit calculation, and likelihood scans. The library takes a modular approach, allowing users to include some or all of its functionality in their workflow.

1 Introduction

Statistical inference in High Energy Physics (HEP) is frequently performed with template fits. In the case of binned templates, the required probability density functions (pdfs) are obtained from a structure of histograms. Models used at the Large Hadron Collider (LHC) can require tens of thousands of histograms or more; those histograms describe nominal distributions as well as systematic variations for observables in different phase space regions and for various different processes. A popular and frequently used statistical model is `HistFactory` [1]. This model defines a prescription for constructing pdfs from a small set of building blocks. The resulting pdfs are sufficiently general to cover a wide range of HEP use cases.

Each `HistFactory` model can be serialized to a so-called workspace. The workspace defines the associated pdf, and `HistFactory` together with `RooFit` [2] can translate a workspace into a pdf. The `pyhf` [3, 4] Python library implements a `JSON` version of the `HistFactory` workspace, and can also convert that into the associated pdf. It furthermore includes utilities to translate between the `ROOT` [5] and `JSON` versions of a workspace, which contain identical information.

Unbinned models are also used in HEP, and libraries like `RooFit` handle both binned and unbinned cases. The `cabinetry` [6] library described in this document deals with binned template fits. Since the two types of models are made up of different building blocks, the restriction to binned fits allows for a more targeted interface design and a restricted scope.

[*]e-mail: alexander.held@nyu.edu

```
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: "353QH, twice"
    Samples: "Background"
    Type: "NormPlusShape"

NormFactors:
  - Name: "Signal_norm"
    Samples: "Signal"
    Nominal: 1
    Bounds: [0, 10]
```

Listing 1: Example of a `cabinetry` configuration file.

The `cabinetry` library provides a Python-based solution for building `HistFactory` models and steering statistical inference with them. A modular design allows users to benefit from the functionality they need, without requiring them to fully commit to `cabinetry` for every step in their statistical analysis. A workspace is like a cabinet: it organizes data in many bins, similar to drawers in a cabinet. The construction of such workspace "cabinets" is *cabinetry* and gives the library its name.

The `HistFitter` [7] package provides functionality for handling `HistFactory` models similar to `cabinetry`. In contrast to `HistFitter`, `cabinetry` uses a declarative approach for defining a fit model. While `HistFitter` is based on the `ROOT` ecosystem, `cabinetry` uses the scientific Python ecosystem, and in particular makes heavy use of the Python implementation of the `HistFactory` model provided by `pyhf`.

2 A declarative configuration for model definition

The creation of statistical models with `cabinetry` is steered by a declarative configuration. A configuration example, serialized to `YAML`, is shown in listing 1. The serialization to `YAML` is chosen here for its increased readability. Users may equally provide configuration files in `JSON` format, or directly use a Python dictionary. The configuration contains multiple blocks:

- `General` contains global settings. This includes a template for the path to data from which histograms can be built, and the path to the folder in which histograms will be saved. The parameter of interest (POI) is also defined here, it is required for a valid `HistFactory` workspace.

- Regions is a list of phase space regions (also referred to as *channels* in HistFactory). A typical analysis contains multiple such regions, each with a unique name. The Filter specifies the selection that is applied to the input. Histograms for each region are binned in an observable that is provided alongside the binning to be used.
- Samples is a list of different processes expected to contribute to an analysis, as well as the observed data. Each sample is identified by a unique name, and a flag is used to differentiate observed data from other samples. The SamplePaths values are combined with the InputPath setting of the General block to obtain the full path to a file. The default implementation for building histograms in cabinetry reads data from ROOT files, and the Tree value specifies the tree containing the events for a sample. Event weights used for histogram filling are also listed here, and can differ between samples.
- Systematics contains a list of settings defining systematic uncertainties. Each systematic uncertainty can be identified with a unique name, and acts on one or multiple samples. Systematics with Normalization type only change the normalization of samples (resulting in a normsys modifier in the pyhf workspace). The NormPlusShape type implements correlated normalization and shape effects (additionally creating a histosys modifier with pyhf). Systematic variations that do not modify the shape of a distribution, like the luminosity uncertainty in the example, are fully specified without requiring the generation of additional template histograms. Other systematic variations require the creation of template histograms, which are called *up* and *down* variations. The HistFactory model specifies how to interpolate between and extrapolate beyond these templates to provide a pdf for any parameter value. When building histograms for these variations, the settings specified for the variations override the nominal settings (obtained from the respective Regions and Samples blocks). It is possible to define additional modifications to apply to template histograms for systematic variations, including symmetrization and smoothing.
- NormFactors is a list of normalization factors that scale the normalization of samples. A normalization factor can be applied to one or multiple samples, and its nominal value and boundaries (to be used when performing fits) can also be defined.

The structure of regions, samples and systematics defines all template histograms needed to build a HistFactory statistical model from a cabinetry configuration. It minimizes duplication of information to retain readability of configuration files for complex statistical models. While the example in listing 1 is shown in YAML format, the cabinetry configuration does not make use of any features outside the JSON specification. The required configuration structure is specified by a JSON schema, which allows for static validation of configurations.

The configuration only contains the minimum information required for building a pyhf workspace. It defines the statistical model, but not the inference steps to be performed with the model. The cabinetry Python API and command line interface (CLI) are used instead to steer statistical inference. This ensures a consistent user experience regardless of whether an analyzer has built a workspace with cabinetry or obtained a workspace in another fashion.

3 Workspace building

Workspaces are built with cabinetry in three steps:

- template histograms are created from columnar data,
- optional post-processing such as smoothing is applied to templates,
- the templates are assembled into a HistFactory workspace.

cabinetry parses a configuration to determine which template histograms are required, and simultaneously generates instructions for constructing each template from columnar data. The instructions are carried out by a backend, which receives histogram building instructions and returns a histogram. The default backend implementation makes use of uproot [8] and boost-histogram [9] to create histograms from columnar data provided in ROOT format. The boost-histogram library is a core dependency of cabinetry alongside Awkward Array [10]. In contrast to this, uproot is an optional dependency: it is only required when using cabinetry to create histograms with the uproot backend. A coffea [11] backend is being developed at the time of writing this document, and a further expansion of backends is planned in the future.

An optional post-processing step can be performed next. This step produces new versions of template histograms from the histograms produced in the previous step. An example operation to apply is template smoothing; this can be useful to avoid spurious constraints originating from templates generated from a small number of simulated events and therefore subject to large statistical fluctuations.

With all required templates produced, cabinetry constructs a HistFactory workspace, which can be serialized to JSON (and optionally converted to ROOT format via pyhf). The workspace combines information from the template histograms produced in the previous steps and the cabinetry configuration.

4 Inference with cabinetry

cabinetry provides a range of utilities to study fit models and perform statistical inference. This functionality can be used with any HistFactory workspace, regardless of whether it was built with cabinetry, as described in section 3, or provided another way. No cabinetry configuration is required for the following features either, since all relevant information is included in the workspace or provided via the Python API or CLI.

Beyond maximum likelihood fits, cabinetry includes interfaces for the following:

- discovery significance calculation for positive signals,
- calculation of observed and expected upper parameter limits,
- likelihood scans, where maximum likelihood fits are performed with one parameter held fixed at various values within an interval,
- ranking of nuisance parameters by their impact on the POI.

Figure 1 shows two examples of visualizations available for the results of maximum likelihood fits. The left part of the figure shows the best-fit results $\hat{\theta} \pm \Delta\theta$ for nuisance parameters. Their nominal parameter values $\theta_0 = 0$ are specified by the HistFactory model. The right part of the figure visualizes correlations between parameters. In this example, parameters are only included in the figure if the magnitude of their correlation with any other parameter is at least 20%.

Figure 2 (left) shows an example of a visualization of a model and its associated uncertainties after being fit to data. This figure requires some metadata not present in the HistFactory workspace; neither bin edges nor the name of the observable are stored. cabinetry falls back to default values when producing such a figure, but the corresponding metadata for bin edges and the horizontal axis label can also be provided. This can be useful to visualize the contents of unknown workspaces. Figure 2 (right) shows a likelihood scan performed with cabinetry. The likelihood offset in each fit compared to the global maximum likelihood is visualized and compared to a Gaussian approximation taken from the covariance matrix at the global minimum.

https://doi.org/10.1051/epjconf/202125103067

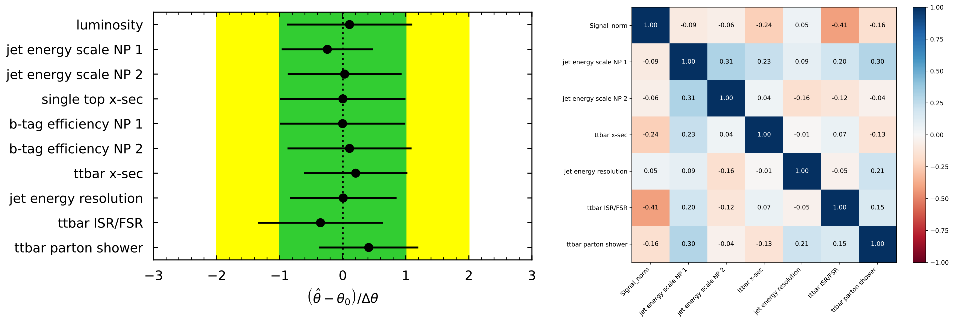

Figure 1. Visualizations for the result of a maximum likelihood fit: best-fit results for nuisance parameters (left) and correlations between parameters (right). A toy model is used for demonstration purposes.

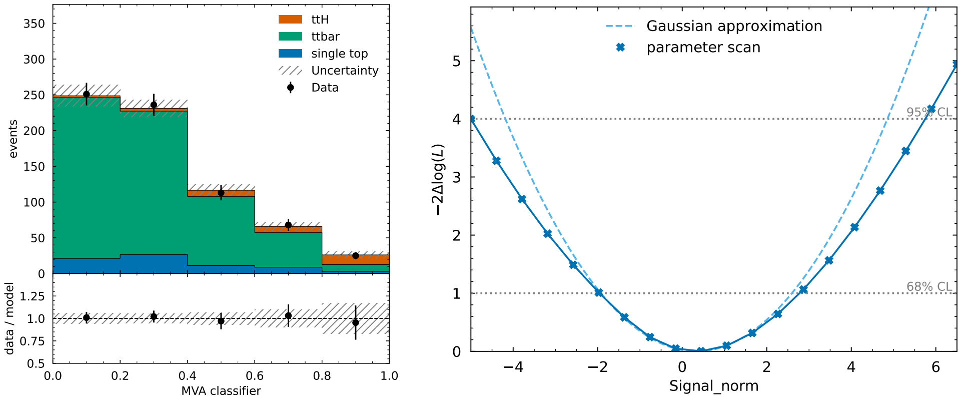

Figure 2. Model predictions and associated uncertainties after a fit to data (left), results of a likelihood scan (right). A toy model is used for demonstration purposes.

Figure 3 (left) visualizes both observed and expected $\mathrm{CL_S}$ [12] values as a function of the POI. The POI value at which the $\mathrm{CL_S} = 5\%$ crossing happens is determined with Brent bracketing [13]. Figure 3 (right) shows nuisance parameters ranked by their impact on the POI. The impact of a nuisance parameter θ is defined as the shift in POI value between different fits. It is given by the difference between the nominal maximum likelihood estimate of the POI, and the POI values in fits where the nuisance parameter θ is held fixed at values $\hat{\theta} \pm \Delta\theta$ (pre-fit impact in *up* and *down* directions) and $\hat{\theta} \pm \Delta\hat{\theta}$ (post-fit impact in *up* and *down* directions).

5 Python API and command line interface

Interactions with cabinetry happen through a Python API or a CLI. An example of using cabinetry via the Python API is shown in listing 2. The configuration steers the first three steps required to build a workspace as described in section 3. Template histograms are generated for the model specified in a configuration. Following a post-processing step, a

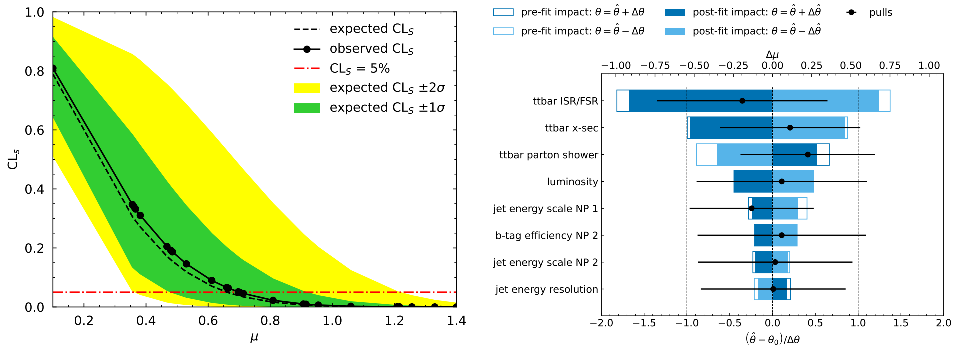

Figure 3. Brent bracketing to determine 95% confidence level upper parameter limits (left), nuisance parameters ranked by their impact on the POI (right). A toy model is used for demonstration purposes.

```
import cabinetry

config = cabinetry.configuration.load("config.yml")

# create template histograms
cabinetry.template_builder.create_histograms(config, method="uproot")

# perform histogram post-processing
cabinetry.template_postprocessor.run(config)

# build a workspace
ws = cabinetry.workspace.build(config)

# fit for maximum likelihood estimate
model, data = cabinetry.model_utils.model_and_data(ws)
fit_results = cabinetry.fit.fit(model, data)

# visualize pulls and correlation matrix
cabinetry.visualize.pulls(fit_results)
cabinetry.visualize.correlation_matrix(fit_results, pruning_threshold=0.1)

# visualize the post-fit model prediction and data
cabinetry.visualize.data_MC(model, data, fit_results=fit_results)
```

Listing 2: Example of using `cabinetry` to build a workspace, perform a maximum likelihood fit and visualize the results.

`HistFactory` workspace is built. The workspace defines the pdf (called `model` here) and the data that this model is fit to. `cabinetry` stores the results of statistical inference, such as the maximum likelihood fit performed in this example, in lightweight containers. The example concludes with a visualization of fit results and of the model prediction after the fit to data, resulting in figures like figure 1 and figure 2 (left).

The CLI provides access to core functionality of `cabinetry` with more limited control. Its immediacy allows for quick exploration of unknown workspaces. The example in listing 3 uses `cabinetry` to perform steps similar to listing 2: creation of template histograms, generation of a workspace after histogram post-processing, maximum likelihood fit with visualization of fit results.

```
$ cabinetry templates config.yml
$ cabinetry postprocess config.yml
$ cabinetry workspace config.yml workspace.json
$ cabinetry fit --pulls --corrmat workspace.json
```

Listing 3: Example of building a workspace and performing a maximum likelihood fit with the `cabinetry` CLI.

```python
import boost_histogram as bh
import cabinetry
import numpy as np

my_router = cabinetry.route.Router()

# define a custom template builder function that is executed for data samples
@my_router.register_template_builder(sample_name="Data")
def build_data_hist(
    region: dict, sample: dict, systematic: dict, template: str
) -> bh.Histogram:
    hist = bh.Histogram(
        bh.axis.Variable(region["Binning"], underflow=False, overflow=False),
        storage=bh.storage.Weight(),
    )
    yields = np.asarray([17, 12, 25, 20])
    variance = np.asarray([1.5, 1.2, 1.8, 1.6])
    hist[...] = np.stack([yields, variance], axis=-1)
    return hist

config = cabinetry.configuration.load("config.yml")
cabinetry.template_builder.create_histograms(config, router=my_router)
```

Listing 4: Example of a custom function providing a histogram to `cabinetry`.

6 Beyond declarative specification

While the declarative configuration approach used by `cabinetry` to specify template building instructions is meant to support a wide range of use cases, additional flexibility may sometimes be required. The limitations of the declarative approach can be circumvented via a mechanism that allows execution of custom code provided to `cabinetry`. Users can write functions that receive dictionaries containing the information about the template being processed, and then register them to `cabinetry` via a decorator. `cabinetry` will call these functions for all templates that match the pattern in the provided decorator. An arbitrary amount of these custom functions can be supplied in this fashion, and they can be executed for different phase space regions, samples, or systematic variations. When no user-provided function is available for a given template histogram, `cabinetry` falls back to its default implementation of sending the template building instructions to a backend for execution.

An example of the mechanism is shown in listing 4. The custom function is executed for data samples with appropriate sample name and returns a histogram. More complex functionality can be implemented by using the information provided via the function arguments.

7 Future directions

The version of `cabinetry` described in this document delivers workspaces in `HistFactory` format. A single parameter is varied up and down within its uncertainty from an auxiliary measurement to obtain the `HistFactory` template histograms used to interpolate and extrapolate systematic uncertainties.

One goal of future versions of `cabinetry` is to support a wider range of interpolations between template histograms. The most general case for this are template histograms obtained when varying an arbitrary set of nuisance parameters by an arbitrary amount. A second goal is to make model construction with `cabinetry` differentiable. The `neos` [14] project demonstrates how automatic differentiation through multiple steps in a HEP workflow can be used to optimize the sensitivity of an analysis via gradient descent. Improvements to the visualization API of `cabinetry` are a short term goal, aiming to allow users to further customize figures and enable the use of `mplhep` [15] for styling.

8 Conclusion

`cabinetry` is a Python library for building and steering binned template fits. It makes use of the scientific Python ecosystem, and in particular of `pyhf`, to deliver its functionality. Statistical models are defined in a declarative manner, but `cabinetry` allows breaking out of the declarative system to implement custom functionality when needed. A Python API and a CLI can be used to steer model building and statistical inference. `cabinetry` implements many frequently used calculations and diagnostics to study and disseminate fit results. The included visualization tools can be used to both quickly learn more about unknown workspaces, and to study complex fits in detail. With its modular design, analyzers are free to choose whether they only want to use a subset of the features provided by `cabinetry`, or make use of the full library.

This work was supported by the U.S. National Science Foundation (NSF) Cooperative Agreement OAC-1836650 (IRIS-HEP).

References

[1] K. Cranmer, G. Lewis, L. Moneta, A. Shibata, W. Verkerke, Tech. Rep. CERN-OPEN-2012-016 (2012), `https://cds.cern.ch/record/1456844`

[2] W. Verkerke, D.P. Kirkby, eConf **C0303241**, MOLT007 (2003), `physics/0306116`

[3] L. Heinrich, M. Feickert, G. Stark, *pyhf* (2021), `https://doi.org/10.5281/zenodo.1169739`

[4] L. Heinrich, M. Feickert, G. Stark, K. Cranmer, Journal of Open Source Software **6**, 2823 (2021)

[5] R. Brun, F. Rademakers, Nucl. Instrum. Meth. A **389**, 81 (1997)

[6] A. Held, M. Feickert, *cabinetry* (2021), `https://doi.org/10.5281/zenodo.4742752`

[7] M. Baak, G.J. Besjes, D. Côte, A. Koutsman, J. Lorenz, D. Short, Eur. Phys. J. C **75**, 153 (2015), `1410.1280`

[8] J. Pivarski, H. Schreiner, C. Burr, D. Davis, N. Hartmann, A. Novak, C. Rappold, G. Stark, J. Bendavid, M. Peresano et al., *Uproot* (2021), `https://doi.org/10.5281/zenodo.4340632`

[9] H. Schreiner, H. Dembinski, N!no, C. Maji, C. Burr, D. Davis, K. Gizdov, P. Grimaud, *boost-histogram* (2021), `https://doi.org/10.5281/zenodo.3492034`

[10] J. Pivarski, P. Das, I. Osborne, A. Biswas, N. Smith, H. Schreiner, N. Hartmann, D. Kalinkin, C. Burr, L. Gray et al., *Awkward Array* (2021), `https://doi.org/10.5281/zenodo.4341376`

[11] L. Gray, N. Smith, A. Novak, D. Thain, D. Taylor, P. Fackeldey, C. Carballo, P. Gessinger, J. Pata, D. Kondratyev et al., *coffea* (2021), `https://doi.org/10.5281/zenodo.3266454`

[12] A.L. Read, Journal of Physics G: Nuclear and Particle Physics **28**, 2693 (2002)

[13] R.P. Brent, *Algorithms for minimization without derivatives* (Prentice-Hall, Englewood Cliffs, NJ, 1973)

[14] L. Heinrich, N. Simpson, *neos* (2020), `https://doi.org/10.5281/zenodo.3697980`

[15] A. Novak, M. Feickert, H. Schreiner, J. Eschle, M. Marinangeli, N!no, Andreas, A. Weiden, L. Gray, B.V.S. Narayanan et al., *mplhep* (2021), `https://doi.org/10.5281/zenodo.4540424`