

(CMS) metadata in the bamboo framework

Pieter David Sébastien Wertz

Université catholique de Louvain

HSF Data Analysis working group: metadata discussions
16 February 2021



- an analysis framework based on RDataFrame, in python
- used in several CMS analyses that use NanoAOD (centrally produced analysis tree format aiming to cover $> 50\%$ of CMS analyses, details)

Depending on the perspective:

- a set of tools to efficiently build RDF graphs (JIT-compiled)
- an embedded domain-specific language for producing plots, skims *etc.* (compact declarative code, effectively an analysis description language)

Design principles and goals:

- ❶ avoid the black box effect: the user makes all the choices that affect the results (using the helpers and building blocks that are provided)
- ❷ analysis code should be as simple and compact as possible (given 1)
- ❸ be as fast as possible (main target: turnaround time on a batch system)

More general overview: HSF workshop; more technical: ROOT PPPM

Building an analysis with bamboo: code

- build expressions from the decorated NanoAOD and helper functions

```
muons = op.select(t.Muon, lambda mu : op.AND(mu.mediumId,  
    mu.pfRelIso03_all < 0.4, mu.pt > 15., op.abs(mu.eta) < 2.4))  
dimu = op.combine(muons, N=2, lambda m1,m2 : m1.charge != m2.charge)  
l1 = dimu[0]  
m_ll = (l1[0].p4 + l1[1].p4).M()
```

- cut flow: define Selection objects by adding cuts and weights to the parent selection (starting from all events in the input, weight 1)

```
if isMC(sample):  
    baseSel = baseSel.refine("mcWeight", weight=t.genWeight)  
diMuSel = baseSel.refine("hasDimu", cut=( op.rng_len(dimu) >= 1 ))
```

- plots: construct Plot objects from axis variable(s) and a Selection

```
plot = Plot.make1D("dimuM", m_ll, diMuSel, EqB(100, 20., 120.))
```

- wrap this up in a module

```
class DimuonPlots(NanoAODHistoModule):  
    def definePlots(self, evt, noSel, sample=None, sampleCfg=None):  
        ## analysis code  
        return plots ## list of Plot objects
```

From this code to stack plots

```
bambooRun -m dimu.py:DimuonPlots dimu_example.yml -o test_dimu1
```

- Samples (input files, name, scaling) are defined in a YAML file, e.g.

`samples:`

`DY_M10to50_2017:`

`group: DY`

`era: "2017"`

`db: "das:/DYJetsToLL_M-10to50_TuneCP5_13TeV-madgraphMLM-pythia8/..."`

`cross-section: 18610`

`generated-events: 'genEventSumw'`

`split: 2`

- `definePlots` gets all sample metadata, so can adjust the graph (scalefactors, corrections *etc.*); it is only called once per sample (or batch job) to construct the RDF graph, which then runs at compiled speed
- by default combined in a stack plot with plotIt, but easy to customize
- Command-line arguments: restrict era, run on batch system, enable implicit multi-threading, verbose logging, re-run only plotting step, ...

Many kinds of metadata

- Main idea: analysis module and YAML configuration file contain all analysis choices (no defaults in the framework), and should be kept under version control. Shared modules and references to other sources are used where appropriate.
- Analysts need flexibility, so provide options and allow for customisation.
- What goes into the YAML or directly in the python code is often a matter of taste and convenience, *e.g.* correction files that are the same for all MC samples of a data-taking year are usually specified directly in the code
- There is a hook to modify the overall config after loading, which analyses can use to *e.g.* select some samples (and avoid having multiple config files), or even generate most of it from a smaller, more compact, format
- Currently many files with weights and corrections are kept in the analysis repositories, but many are shared, so could be stored on CMVFS or in a database

Specifying datasets

`samples` block of the YAML configuration file

- Mandatory: unique name, file names (preferably a database link or query, otherwise path of a textfile with list of paths, or listed directly)
- MC normalisation: sum of weights calculated automatically (in-file metadata), cross-section (per sample) and luminosity (per era) in YAML. Most common choice: store unscaled histograms, scale when plotting.
- When using multiple eras (e.g. data-taking years): specify era per sample for per-era processing and plots
- For data: run range and data quality file (JSON from CMS certification)
- The full dictionary is passed to `definePlots`, so trivial to pass additional information, e.g. about specific corrections
- Feedback from users: transparent but rather verbose, so files may get large. Can be mitigated using hook (e.g. duplicate sample across eras)
- Idea for future developments: integration with tools for workflow management and reproducible analysis platforms (REANA *etc.*)

Corrections and weights files

- Scale factors (efficiency ratio or weight maps): JSON files
 - Currently: simple format from our previous framework (conversion scripts from ROOT histograms); near future: a new format shared across CMS
 - More self-contained than ROOT files with histograms, and a single C++ reader can be used for all scale factors: analysis code only needs to call a helper method with the path of the JSON file
 - Used for lepton efficiencies, pileup weights, b-tagging efficiencies (together with a standalone class from the b-tagging group to read scale factors) *etc.*
- JEC/JER variations: textfiles from CMS jet&MET reconstruction group
 - Only if these are done on-demand, can also use “postprocessed” NanoAOD, which has these variations stored as additional branches
 - Text representation of conditions database payloads
 - Reader classes from CMSSW (copied and patched for standalone build)
 - On GitHub, > 1 GB repository for JEC, the few files that are needed are downloaded automatically through the GitHub API and cached locally
 - Future: shared CMS JSON format and common store
- MVA classifiers: weights files passed to readers that use the C(++) APIs for TMVA, Tensorflow, PyTorch (TorchScript), lwttn, and ONNX-Runtime

Is this also metadata?

- Systematic uncertainties are propagated automatically, but need to be enabled in the code (e.g. passing a variation name when loading a scale factor, when configuring jet and MET on-demand calculators, or setting up the decoration of postprocessed NanoAOD), and (currently, could be automated) a list specified of which to include in the plots
- Data-driven backgrounds: done by 1) specifying (in the YAML) which samples or sample groups (e.g. data) should be used to estimate the background and which should be replaced with the result, and by 2) defining a special `Selection` object in the code ([example](#))
- Options for plot layout: defaults in YAML, per-plot customisation from python code (whole plot definition in one place)

It is not always possible (or practical) to fully separate code, configuration, and metadata e.g. how to run, locally or on a batch system, does not change the results, so is a command-line argument, but the per-sample splitting in batch jobs is specified in the YAML config

Conclusions

- Introduced the bamboo framework, and described handling of metadata
- Current approach: provide tools to cover all use cases, but reuse where possible (e.g. JSON format or existing code)
- The shared CMS JSON format that is currently under development should allow to simplify some things, and directly use more corrections that are produced and stored centrally
- Many needs are shared across analyses, but specifics can strongly differ — this is an interesting topic, looking forward to hearing your ideas and the discussion!

Additional material

Bamboo JSON scale factor format

```
{
  "dimension": 2,
  "variables": [
    "AbsEta",
    "pT"
  ],
  "binning": {
    "x": [
      20.0,
      25.0,
      30.0,
      40.0,
      50.0,
      60.0,
      120.0
    ],
    "y": [
      0.0,
      0.9,
      1.2,
      2.1
    ]
  },
  "data": [
    {
      "bin": [
        20.0,
        25.0
      ],
      "values": [
        {
          "bin": [
            0.0,
            0.9
          ],
          "value": 0.991608016077216,
          "error_low": 0.0038204177782513350,
          "error_high": 0.0038204177782513350
        },
        {
          "bin": [
            0.9,
            1.2
          ],
          "value": 1.008734907719776,
          "error_low": 0.006698366742712355,
          "error_high": 0.006698366742712355
        }
      ]
    }
  ]
}
```

- Simple and largely self-documenting JSON format
- Principle: N-dimensional histogram, with the option to specify functions instead of values
- C++ reader (using TFormula for formulas)
- Simple python conversion scripts from ROOT files to JSON, some examples in this directory
- Limitations ($N \leq 3$, square bins, systematic variations) to be addressed by common CMS JSON format

Example YAML file with user hooks

```
samples:
  TTTo2L2Nu_hdampUP_TuneCP5_13TeV-powheg-pythia8:
    regex: "TTTo2L2Nu_hdampUP_TuneCP5(_PSweights)?_..._(.ext.)?"
    eras: ['2016', '2017', '2018']
    cross-section: *xs_tt_2l
    is_signal: true
    syst: ['hdampup', 'TTTo2L2Nu_TuneCP5_13TeV-powheg-pythia8']
    generated-events: genEventSumw
```

- Automatic duplication of sample across eras, fetch sample storage path using regex
- The sample is used to estimate systematic uncertainties: specify the corresponding nominal sample and the uncertainty
- Flag the sample as part of the signal process
- Use YAML reference to pick up cross section from nominal sample (defined elsewhere as cross-section: &xs_tt_2l 91.5)