Cascades Combine Framework Documentation

Zach Flowers

Combine Overview

- Made fork of LLPCombine repo from Justin presented here

- Updated to work with Cascades skims
    - Creates both JSON yields (to be fed into combine for limits) and histograms of various types

- Link to repo: https://github.com/zflowers/CascadesCombine

BuildFitInput Overview

- Conceptually the workflow is
  - BuildFitInput (BFI) -> BuildFit (BF) -> Combine
- The output of each step is
  - JSON yields/bin -> datacards -> limits
- To more efficiently run over multiple datasets we parallelize BFI processing using HTCondor at the LPC
    - A checker is implemented to catch failed jobs
- Total time to process from launching BFI to limits is ~5-15 minutes depending on number of bins and assuming not also making histograms

Framework Installation

- README has detailed instructions for installing the framework and necessary subpackages (Combine, CombineHarvester, etc.)
  - Use the LPC and the EL9 nodes to install and run things!
    - Recommended to install in your ~/nobackup/ area as ~/$HOME/ does not have enough space
- I included an example .bash_profile which has useful aliases to use for setting up the framework after installing it

Running The Framework

- Many helper scripts and executables but main script to run is python/run_combine.py
    - Calls all necessary commands to go from bin definitions to final significance
- YAML config files are used to load processes, bins, and if desired, histograms
- Also this is where the integrated luminosity is set (default to 400)
    - In the future, SampleTool will be updated to properly handle by year lumis

Dataset Lists

- YAML config file specifies which processes to load
  - Example process names: ttbar, QCD, DY, ZInv…
- Dataset paths linked to processes defined in SampleTool.cc
  - End users should not need to edit but useful if you want to look up specifically which datasets correspond to which processes
- Backgrounds are easy to add
- Cascade signals: creates a key based on the stored masses in the first event
- SMS signals: creates key based on masses from tree name
- For both types of signals, framework keys in on “SMS” or “Cascades” to know what to do

Bins

- A “bin” is a region of phase space (kinematic, objects, etc.) defined by given cuts
- Bin definitions can be very simple (ex: 2 leptons and 150 GeV of MET) or more complicated
- For every bin inputted into the framework, combine will add that bin to a simultaneous fit
- Bins are loaded from .yaml files in config/bin_cfgs/

Defining Bin Cuts

- Four different “types” of cuts: “Square”, “Predefined”, “LeptonCuts”, “UserCuts"
- “Square”: cuts directly on stored branches: “MET>150”
- “Predefined”: Stored as members of BFI for easier use by user instead of typing out full string (“Cleaning”)
- “LeptonCuts”: Uses regex matching to form cuts based on flavor, charge, and pair combinations (OSSF)
- “UserCuts”: Users can define their own custom columns to cut on as well
  - Multiple examples provided in BuildFitInput::loadCutsUser()

Lepton Cuts

- “LeptonCuts”: Uses regex matching to form cuts based on flavor, charge, and pair combinations (OSSF)
    - Can take RJR hemisphere assignments
        - Add “_side” so for example: “=2Pos” -> “=2Pos_a” means two positive leptons on “A” side
    - All skim RJR trees for both object counting and kinematics use convention that “A” side is the side with the larger visible invariant mass
- Can also do mass (veto or cut) or DeltaR ranges on lepton pairs
    - Example J/Psi veto and mass < 65 GeV:
        - =1OSSF|mass![3.,3.2]|mass<65;

Examples

```
shorthands = [
      "=0Bronze",
      "=2Pos",
      "=2Gold",
      ">=1OSSF",
      "=1SSOF",
      ">=2Mu",
      ">=1Elec",
      "<1SSSF",
      ">=1Muon"
]
```

More examples in this config

User Cuts

- “UserCuts” are defined in BuildFitInput.cpp
- As part of the definition, the user gives a name for the cut
- This name can then be added to the bin definition in their .yaml under ‘user-cuts:’
- Can string together multiple user cuts with ‘;’

Define column “M_jj” and what it should return

Give cut a name to be used in .yaml “M_jj_gt_100”

Tell the cut which columns to use (that you just defined)

Provide the expression, so M_jj > 100 would mean keep events with M_jj > 100

Link the name of the cut to full CutDef

```
// Example 1: invariant mass of leading two jets -> M_jj (double)
node = node.Define("M_jj", [](const std::vector<double> &pt,
                              const std::vector<double> &eta,
                              const std::vector<double> &phi,
                              const std::vector<double> &mass) {
    // Return -1.0 if not enough jets
    if (pt.size() < 2 || eta.size() < 2 || phi.size() < 2 || mass.size() < 2) return -1.0;
    // compute using temporary TLorentzVector locally (we only return a double)
    TLorentzVector j0, j1;
    j0.SetPtEtaPhiM(pt[0], eta[0], phi[0], mass[0]);
    j1.SetPtEtaPhiM(pt[1], eta[1], phi[1], mass[1]);
    return (j0 + j1).M();
}, {"PT_jet","Eta_jet","Phi_jet","M_jet"});

CutDef cut1;
cut1.name = "M_jj_gt_100";
cut1.columns = {"M_jj"};
cut1.expression = "M_jj > 100";
cuts[cut1.name] = cut1;
```

Output Types

- The run_combine.py flags --make-root and --make-json are used to turn on whether to store histogram output in a root file and/or yields in a JSON

- If a user passes neither, the JSON output is saved by default (otherwise the jobs would save nothing)

- run_combine.py automatically handles merging sub-output json and root files for later steps

Defining Histograms

- Histogram definitions are primarily loaded with a .yaml
- There is the DefineUserHists.h file which can also be used to define histograms in the same format
  - Gives user a bit more control over what the histogram should be filled with

```yaml
- name: RISR_vs_PTISR
  type: "2D"
  expr: "RISR"
  yexpr: "PTISR"
  nbins: 64
  xmin: 0
  xmax: 1
  nybins: 20
  ymin: 250
  ymax: 800
  x_title: "R_{ISR}"
  y_title: "p_{T}^{ISR}"
  cuts: ""
  lep-cuts: ""
  predefined-cuts: ""
  user-cuts: ""
```

Defining Histograms

- Can also add additional cuts for that specific histogram and accepts all type cuts defined earlier

- Can also define your own variable expressions to plot

Two different Mperp histograms with different RISR ranges

```yaml
- name: Mperp_RISRHigh
  type: "1D"
  expr: "Mperp"
  nbins: 64
  xmin: 0
  xmax: 150
  x_title: "M_{#perp} for R_{ISR}>=0.85"
  cuts: "RISR>=0.85"
  lep-cuts: ""
  predefined-cuts: ""
  user-cuts: ""

- name: Mperp_RISRLow
  type: "1D"
  expr: "Mperp"
  nbins: 64
  xmin: 0
  xmax: 150
  x_title: "M_{#perp} for R_{ISR}<0.85"
  cuts: "RISR<0.85"
  lep-cuts: ""
  predefined-cuts: ""
  user-cuts: ""
```

Custom variables based on branches already in the tree. SafeDiv and SafeIndex are helpers to avoid /0 and index out of bounds

```yaml
derived_variables:
  - name: MV_ratio
    expr: "SafeDiv(MVa, MVb, 0.0)"
  - name: PT3_lep
    expr: "SafeIndex(PT_lep, 2, -1.0)"
```

Add “derived variables” can then be plotted on their own, vs each other, or vs other quantities already in the tree

```yaml
- name: MV_ratio_vs_PT3
  type: "2D"
  expr: "MV_ratio"    # X-axis
  yexpr: "PT3_lep"    # Y-axis
  nbins: 20
  xmin: 1
  xmax: 2
  nybins: 50
  ymin: 0
  ymax: 50
  x_title: "MVa/MVb"
  y_title: "p_{T}^{3l}"
  cuts: ""
  lep-cuts: ""
  predefined-cuts: ""
  user-cuts: ""
```

***Derived variables not used for making cuts**

Defining User Histograms

- There is the DefineUserHists.h file which can also be used to define histograms in the same format
  - Here you can define new columns to fill histograms with too
- Note that if you use both the DefineUserHists and the loadCutsUser to define new columns, you need to give the columns different names

User histograms use same fields as YAML

Maintains same functionality for adding additional cuts

```
// 2) 2D histogram: M_ll (x) vs HTeta24_over_MET (y)
//    This uses the derived ratio defined above
HistDef h2;
h2.name = "M_ll_lead2_vs_HTeta24overMET";
h2.type = "2D";
h2.expr = "M_ll";               // X-axis: invariant mass of leading two leptons
h2.yexpr = "HTeta24_over_MET";   // Y-axis: derived HT/MET ratio
h2.nbins = 50;
h2.xmin = 0;
h2.xmax = 100;
h2.nybins = 50;
h2.ymin = 0;
h2.ymax = 3;
h2.x_title = "M_{ll} for OSSF pair of lead leps";
h2.y_title = "HT/MET";
h2.cuts = {};       // no extra event-level cuts here (hist loop may also apply global cuts)
h2.lepCuts = {};
h2.predefCuts = {};
h2.userCuts = {};
```

Plotting Histograms

- PlotHistograms.cpp takes the root file output from earlier steps in run_combine (after merging) and makes plots of all stored histograms

- Automatically plots 1D, 2D, and 1D Cut Flows for each bin

  - Can also make efficiencies if user defines numerator and denominator histograms with the conventions “num__” and “den__” (note that’s two underscores)

    - For 1D histograms, it will also automatically make stack plots from them

- Also makes 2D yield plots where x-axis is the bins and y-axis is the processes

  - Useful for comparing sensitivity contribution from different bins

```yaml
- name: den__METtrigger # example denominator for TEff
  type: "1D"
  expr: "MET"
  nbins: 32
  xmin: 150
  xmax: 500
  x_title: "MET"
  y_title: "METtrig Eff"
  cuts: ""
  lep-cut: ""
  predefined-cuts: ""
  user-cuts: ""
```

```yaml
- name: num__METtrigger # example numerator for TEff
  type: "1D"
  expr: "MET"
  nbins: 32
  xmin: 150
  xmax: 500
  x_title: "MET"
  y_title: "METtrig Eff"
  cuts: "METtrigger==1;"
  lep-cut: ""
  predefined-cuts: ""
  user-cuts: ""
```

Example of defining histograms to be used for making efficiency (MET trigger in this example)

Workflow Summary

- Define your bins, hists, and processes in YAML config files

- Run the run_combine.py script
  - nohup python3 python/run_combine.py --bins-cfg config/mybins.yaml > debug_run_combine.debug 2>&1 &
  - tail -f debug_run_combine.debug

- Look at significances and histograms, and repeat