CascadeCombine 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

Installing The Framework

- 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 install 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 for now 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 processes: ttbar, QCD, DY, ZInv…
- Dataset paths defined in SampleTool.cc
- 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
- End users should not need to edit but useful if you want to look up specifically which datasets correspond to which processes

Defining bins

- Four different “types” of cuts
  - “Square”: cuts directly on stored branches: “MET>150”
  - “Predefined”: Stored as members of BFI for easier use by user (“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 with 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 hemisphere assignments (for now uses old tree/ branch with jets in S)
        - 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

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 ‘;’

Histogram and JSON Outputs

- 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

Histogram Definitions

- Histogram definitions are primarily loaded with a .yaml
- Define the axis labels, axis ranges, column(s) (values to fill) and give it a name
  - Can also add additional cuts for that specific histogram and accepts all type cuts defined earlier
- There is the DefineUserHists.h file which can 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

Histogram Plotter

- 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 cutflows (made for free 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

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

Backup

Upcoming Features (& To Do List)

- New ntuples store weight*weight in addition to just weight — should make things *slightly* more efficient

- Need to update sample paths when new skims are ready — for now everything is using v3

  - Will likely at this stage switch vars to lepton only RJR tree

- Need to think about how to implement reweighing to different BRs

- Working on code to also make histograms along side JSON

  - Useful for isolating remaining discriminating observables all in one place