2D Alphabet Software Documentation

Lucas Corcodilos

June 22, 2018

1 Introduction to the method

As one might expect, the 2D Alphabet background estimation method is a two dimensional version of the Alphabet method. The 1D Alphabet method derives a background estimate in the jet mass signal region by interpolating a pass-to-fail ratio, $R_{P/F}$, from the calculated values in the jet mass sidebands and multiplying the failing distribution in the signal region by this $R_{P/F}$.

This method has been used successfully in the past. However, if the signal region is wide and the statistics in the high jet mass sideband are low, the background estimate is susceptible to shifts that do not accurately describe the behavior of the background being estimated. Additionally, if the $R_{P/F}$ has a strong dependence on the measurement variable (for example, resonance mass), 1D Alphabet will not be able to accurately predict the background in this variable. The 2D Alphabet method attempts to remedy both of these issues by constraining the shape of the $R_{P/F}$ with the measurement variable as the second dimension.

Since the 1D Alphabet method requires a fit of the sideband values to interpolate the signal region, the 2D Alphabet method must do the same but in two dimensions. The statistical tool of choice to accomplish this is the Higgs Analysis Combine Tool. The documentation for the tool can be found here.

2 Prerequisites

2D Alphabet has only been tested in a CMSSW environment. The Higgs Analysis Combine Tool must be installed. Please follow the instructions in the Combine documentation [1] to setup the prerequisite software.

If the command `combine --help` returns the help menu then you've successfully setup the environment (remember to run `cmsenv` before trying this!).

[1] `https://cms-hcomb.gitbooks.io/combine/content/part1/`

3 Installation

PLEASE NOTE THAT THE SOFTWARE IS NOT CURRENTLY READY TO BE INSTALLED AND USED. THIS SECTION IS BEING DEVELOPED ALONG WITH THE SOFTWARE.

Once a CMSSW environment with Combine is setup, one can install the 2D Alphabet software. There are two main steps. The first is to clone the master release from `https://github.com/lcorcodilos/2DAlphabet`. This can be done by doing

- `cd $CMSSW_BASE/src`
- `git clone https://github.com/lcorcodilos/2DAlphabet`

The second step is to copy several files from the 2D Alphabet release to your Combine directory and re-compile Combine so it can use a new class - RooParametricHist2D. You can grab these using the following commands

- `cd $CMSSW_BASE/src/2DAlphabet/setup`
- `source run_setup.csh`

It's a good idea to run `combine --help` again and if you're feeling uneasy, try to call RooParametricHist2D in interactive python to ensure everything is working.

4 How To Run

The 2D Alphabet framework is designed to run by calling a single script (`2DAlphabet.py`) with a JSON configuration file as input via the command line option `-i`. There are several other options that will be discussed later but they are truly "optional" whereas the configuration file is not - so I will discuss that first.

4.1 JSON Configuration

While I've tried to keep the format of the JSON configuration file obvious, it's not fair to assume that everyone else will feel the same way. I've included "HELP" keys in dictionaries where I felt I could easily explain the formatting. This section is a necessary supplement to those. Please note that I may use phrases like, "The key 'HELP' is defined inside 'PROCESS'." I'm aware this is not correct since "PROCESS" is a key for a dictionary which has a key "HELP"

and so "HELP" can't be "inside" "PROCESS" but it simplifies the explanation so I'm going to roll with it.

The goal of the JSON is to have an easily configurable input that allows the 2D Alphabet software to read and organize analysis files and histograms to its liking while also giving the user the ability to easily configure values like bin sizes and ranges. This means that while the user is encouraged to add what they need, there are some keys and values that must stay the same. **These static strings are always in capital letters to make them easy to distinguish.** I'll now describe what each of the five static sections is designed for.

4.1.1 PROCESS

In this section, the user can define as many processes as they need to account for. This includes data and MC. This section is NOT used to define the background to be estimated via the transfer function (typicall QCD). That background is naturally defined by the difference between data and the other backgrounds defined here.

Each key in `PROCESS` is the name of each process of interest. Please name your data as "data_obs." This is a Combine convention that I'd like to maintain. Each process name is a key for a sub-dictionary that specifies

- `FILE:` the path to the file containing the nominal pass and fail histograms for the process (string);
- `HISTPASS:` the name of the histogram in the above file with the passing distribution (string);
- `HISTFAIL:` the name of the histogram in the above file with the failing distribution (string);
- `SYSTEMATICS:` a list of strings that correspond to the names of systematics in `SYSTEMATIC` (note `SYSTEMATIC` not `SYSTEMATICS`) that are applicable to this process (list of strings);
- `CODE:` a way to classify the treatment of the process: 0 (signal), 1 (data), 2 (unchanged MC), 3 (MC to be renormalized) (int).

4.1.2 SYSTEMATIC

Because it bears repeating, please note the difference between this section, `SYSTEMATIC`, and the list of `SYSTEMATICS` defined inside the `PROCESS` dictionary. The `SYSTEMATIC` dictionary is a place to define as many systematics as a user may need. Similar to the processes, each key in `SYSTEMATIC` is the name of the systematic in the analysis and each is classified by a code that determines how the systematic will be treated. However, the dictionary the user defines for a

given systematic is different depending on what type it is. The self-explanatory types are:

- Symmetric, log-normal
  - `CODE: 0` (int)
  - `VAL:` uncertainty (float)
- Asymmetric, log-normal
  - `CODE: 1` (int)
  - `VALUP:` $+1\sigma$ uncertainty (float)
  - `VALDOWN:` -1σ uncertainty (float)

Less obvious are codes 2 and 3 which are for shape based uncertainties (and thus have corresponding histograms) and are either in the same file as the process's nominal histogram (code 2) or in a separate file (code 3). Additionally, they have a scale value which allows the user to change the normalization of the shape. For no change in the normalization, use 1.0. If you have a histogram with a 2 sigma shift, use 0.5 to divide the unit gaussian by 2 before doing the interpolation with Combine.

- Shape based uncertainty, in same file as nominal histogram
  - `CODE: 2` (int)
  - `HISTPASS_UP:` the name of the histogram (in the same file as the nominal histogram) for $+1\sigma$ uncertainty in the pass distribution (string)
  - `HISTPASS_DOWN:` the name of the histogram (in the same file as the nominal histogram) for -1σ uncertainty in the pass distribution (string)
  - `HISTFAIL_UP:` the name of the histogram (in the same file as the nominal histogram) for $+1\sigma$ uncertainty in the fail distribution (string)
  - `HISTFAIL_DOWN:` the name of the histogram (in the same file as the nominal histogram) for -1σ uncertainty in the fail distribution (string)
  - `SCALE:` a scale value which allows the user to change the normalization of the shape (float)
- Shape based uncertainty, in different file as nominal histogram. This is the more flexible but also more complicated option. The user can specify files three different ways. The first is by using `FILEUP:` and `FILEDOWN:` to pick a file that *every* process can pull the shape uncertainty histograms from. The second way is to use keys of the form `FILEUP_myprocess:` where `myprocess` matches the name of a process that is defined in the `PROCESS`

dictionary and has this shape uncertainty associated with it. This allows each systematic and process to come from a separate file. The third way is to use keys of the form `FILEUP_*:` where the * acts as a wild card for the process and must also exist in the file name where the process would normally be written.

For example, if my ttbar distributions with $+1\sigma$ uncertainty are stored in `ttbar_pileup_up.root` and the corresponding signal distribtutions are in `signal_pileup_up.root`, I can use the key value pair `'FILE_UP_*':'*_pileup_up.root'`.

The user can also specify histogram names in four different ways. The first is `HISTPASS` and `HISTFAIL` which allows the user to specify only two histogram names if they don't change between "up" and "down" shapes. The second is if the "up" and "down" shapes *do* have different histogram names and uses the form `HISTPASS_UP` and `HISTFAIL_UP`. Third, the totally generic way allows the user to use the form `HISTPASS_UP_myprocess` where (again) `myprocess` matches the name of a process that is defined in the `PROCESS` dictionary and has this shape uncertainty associated with it. Finally, the "*" wildcard can be used in place of `myprocess` just as with the file keys. Below is an example of the totally generic way.

- `CODE: 3` (int)
- `FILEUP_myprocess: /path/to/fileup_myprocess.root` which contains the $+1\sigma$ uncertainty histogram for myprocess (string)
- `FILEDOWN_myprocess: /path/to/filedown_myprocess.root` which contains the -1σ uncertainty histogram for myprocess (string)
- `HISTPASS_UP_myprocess:` the name of histogram for myprocess in `/path/to/fileup_myprocess.root` for $+1\sigma$ uncertainty in the pass distribution (string)
- `HISTPASS_DOWN_myprocess:` the name of the histogram for myprocess in `/path/to/filedown_myprocess.root` for -1σ uncertainty in the pass distribution (string)
- `HISTFAIL_UP_myprocess:` the name of histogram for myprocess in `/path/to/fileup_myprocess.root` for $+1\sigma$ uncertainty in the fail distribution (string)
- `HISTFAIL_DOWN_myprocess:` the name of the histogram for myprocess in `/path/to/filedown_myprocess.root` for -1σ uncertainty in the fail distribution (string)
- `SCALE:` a scale value which allows the user to change the normalization of the shape (float)

This scheme is quite flexible. However, the more organized you are, the easier it is to write a configuration file.

4.1.3 BINNING

One set of important user defined values is the 2D binning of the space being analyzed. This dictionary is the opportunity to define the axes binning of user's space. The binning values are split into x and y axis definitions where the x-axis describes the variable whose signal region is blinded. Note that the `LOW` and `HIGH` bin edges for the `Y` axis that are defined here *must* be consistent with the user's input histograms. Additionally, the `X` axis binning cannot be beyond the range of the input histogram (hopefully, this is obvious) but it can be a subset of the input range. The number of bins are restricted to be equal to or less than the input number of bins (in both the `X` and `Y` axes) and the signal bounds only apply to the `X` axis and must exist within the `X` axis range defined in the configuration file. Finally, the user must specify if they want to fit their 2D space by blinding the signal region via the `BLINDED` key which can take only boolean `false` and `true`.

- `X`
  - `NAME:` name of your variable on the x-axis (string)
  - `LOW:` lower bound of x-axis (int)
  - `HIGH:` upper bound of x-axis (int)
  - `NBINS:` number of x bins from LOW to HIGH (int)
  - `SIGSTART:` lower bound of signal region of x-axis (int)
  - `SIGEND:` upper bound of signal region of x-axis (int)
  - `BLINDED:` blinds or unblinds the signal region during the $R_{P/F}$ fit (bool)
- `Y`
  - `NAME:` name of your variable on the y-axis (string)
  - `LOW:` lower bound of y-axis (int)
  - `HIGH:` upper bound of y-axis (int)
  - `NBINS:` number of x bins from `LOW` to `HIGH` (int)

4.1.4 FIT

The other set of important user defined values is the fit parameters for the transfer function from the fail sideband to the passing (or pass-fail ratio). The 2D fit of the transfer function assumes a polynomial in both directions. The order of the polynomials are up to the user (technically they are capped at order 10) as long as each parameter is defined. For example, if the highest order parameter defined is 2 in x and 1 in y, the user needs to define six parameters here.

Expanding the number of functions available to the user is currently in development. Because of this, the user is required to write a string for their functional form. The user has the opportunity to use either `FORM` or `XFORM` and `YFORM` to write the formula. `FORM` is used if the x and y terms are mixed. For example, one could use `@1+@2*x+@3*y+@4*x*y` where `@1, @2, @3, @4` would correspond to `X0Y0`, `X1Y0`, `X0Y1`, and `X1Y1` in the configuration file. `XFORM` and `YFORM` are used if the user would like to isolate the behaviors in the x and y variables. For example, the previously formula could instead be written as `(@1+@*2x)` for the `XFORM` and `(@1+@2*y)` for the `YFORM` where `@1, @2` in the `XFORM` would correspond to `X1` and `X2` in the configuration file and `@1, @2` in the `YFORM` would correspond to `Y1` and `Y2` in the configuration file.

There are two conventions here that are important. The first is that the numbering of the parameters in `XFORM` and `YFORM` must start at 1 - not 0. The 0 is reserved for the corresponding variable. This is not the case if `FORM` is used (in fact, fir now the contents of `FORM` are not used). The second convention is that the separate `X` and `Y` parameters start at 1 to match the values in `XFORM` and `YFORM`. However, the parameter number when `FORM` is used starts at 0 (see example below).

Finally, for each parameter, the user must specify `NOMINAL`, `LOW`, and `HIGH` values that correspond to a guess and range for the parameter while it floats in the fit.

An example is given below.

- `FORM:` (@1+@2*x+@3*y+@4*x*y)
- `X0Y0`
  - `NOMINAL:` nominal value (float)
  - `LOW:` lower bound (float)
  - `HIGH:` upper bound (float)
- `X1Y0`
  - `NOMINAL:` nominal value (float)
  - `LOW:` lower bound (float)
  - `HIGH:` upper bound (float)
- `X0Y1`
  - `NOMINAL:` nominal value (float)
  - `LOW:` lower bound (float)
  - `HIGH:` upper bound (float)
- `X1Y1`

  - `NOMINAL:` nominal value (float)
  - `LOW:` lower bound (float)
  - `HIGH:` upper bound (float)
- ...

4.1.5 GLOBAL

This dictionary is designed to help users with large configuration files by allowing them to create JSON-wide variables. For example, if all of your files are located in `/long/path/to/my/variables/`, you can store this string in the GLOBAL dictionary with a custom key (let's say `dir/`). Now instead of having to write the full directory path for every process and systematic, the user can just write `dir/`. This simplifies the JSON and also has the standard advantages of using variables over several instances of the same object.

This works by searching all strings in the JSON for instances of each key in `GLOBAL` and replacing the key with its corresponding dictionary value.

Thus, the user must be careful they don't accidentally use strings in the JSON that are identical to keys in `GLOBAL` but that should be unchanged. This means keys in `GLOBAL` should be descriptive (single character keys would be a bad idea).

4.2 Command line options

There are several command line options that are currently implemented for use when running 2DAlphabet.py. The most important (and required) is `-i` (`--input`) which points to the configuration file which should be named as `input_<tag>.json` where `<tag>` will be the name of the folder where all outputs are saved to keep the user organized.

The remaining options are listed below.

- `-i`, `--input`, JSON file to be imported. Name should be "input_¡tag¿.json" where tag will be used to organize outputs')
- `-s`, `--pseudo2D` Recalculate the fit guesses using pseudo2D method (1D Alphabet in slices)
- `-p`, `--plotOnly`, Only runs the part of the script that is necessary to plot. If you used a command like `-f` when running the full script, that should also be used.
- `-d`, `--draw`, Draws canvases live - for debugging

- `-s`, `--signalOff`, Turns off signal by setting `rMin` and `rMax` options to 0 in Combine
- `-f`, `--runFit`, Runs Combine max likelihood fit and plots outputs
- `-l`, `--runLimits`, Runs Combine limits and plots outputs

5 Plotting

The 2D Alphabet software already makes some basic plots for the user including the shape of the transfer function, several 2D distributions, and the 1D projections onto the y-axis along with the stacked background comparison to data and a pull plot. If you'd like to make more plots from the output of Combine, you'll have to access the RooWorkspaces manually. Most of the time, the workspace that is output by Combine, `MaxLikelihoodFitResult.root`, has the distribution of interest. However, Combine only saves what it deemed useful in its calculations meaning that some things (like distributions in your signal region which Combine is blind to) are not in `MaxLikelihoodFitResult.root`. You can either grab these yourself from the input files referrenced in your JSON or access `base.root` which contains the RooWorkspace input to Combine by the 2D Alphabet software.

To get a better idea of how to do all of this, please take a look at `plot_fit_results.py`.