2D Alphabet Software Documentation

Lucas Corcodilos

June 14, 2019

1 Introduction

2D Alphabet is a wrapper to construct the workspace for a specific type of background estimate, provide input to the Combine Statistical Tool, plot the 2D distributions from the fit result, and provide the infrastructure to test this result.

The name of the wrapper is derived from the data driven background estimate performed to estimate combinatorial backgrounds that are otherwise poorly modeled by Monte Carlo simulation. In many cases, the background being modeled is QCD production and as a default, this is the name of the output background produced in plots. However, depending on the selection, there may be other backgrounds accounted for as well such as V+jets. Please see Section **??** for details on how to change the name of this background from "QCD" to something more appropriate.

As one might expect, the data driven background estimation method is a two dimensional version of the Alphabet method. The 1D Alphabet method works on the concept of a transfer function for non-resonant backgrounds as a function of variable, x, which will be cut on to discriminate signal from background. By defining a descriminator, D, one can separate data events based on whether they pass or fail the selection on D (the sets D_{pass} and D_{fail}. Then resonant backgrounds can be subtracted from these selections to get estimates of the non-resonant background. Since the non-resonant background is smooth in x in both the pass and fail, the pass-to-fail ratio should also be smooth (and positive).

Using this criteria, one can calculate the per-bin ratios as a function of x outside of the signal selection and then fit for a function, $R_{P/F}(x)$, that interpolates the x signal region. Then those events that fail D and exist in the x signal selection can be multiplied by $R_{P/F}(x)$ to estimate D_{pass} in the signal region.

This 1D method has been used successfully in the past. However, if the signal region is wide and the statistics in the x sidebands 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, y, as the second dimension.

While fitting for this surface, it's convenient to also fit for other background contributions. For example, say an analysis has as its main backgrounds QCD and Bkg_A. Bkg_A modeled well enough by simulation that the MC can be used directly. However, there may be a recommendation to fit for some MC-to-data corrections given a systematic uncertainty (for example, this is the recommendation for top p_T reweighting for $t\bar{t}$ MC). This means that the contribution of Bkg_A can change and therefore, the amount subtracted from data to form the initial QCD estimate in the x sidebands can change! Thus, QCD and Bkg_A are fit simultaneously to derive a total background estimate. The details of accounting for systematic uncertainties are explained in Section **??**.

A useful feature of the 2D Alphabet setup is that multiple pairs of pass and fail distributions (called "Alphabet pairs") can be fit simultaneously. For example, one can setup this estimate for 2016, 2017, and 2018 data and selections (three pairs of pass and fail) and then fit all three at the same time with nuisance parameters easily correlated across pairs.

The statistical tool of choice to accomplish this is the Higgs Analysis Combine Tool. The documentation for the tool can be found **here**. One significant addition is made to the central Combine release to make Combine 2D Alphabet-friendly - the RooParametricHist2D class. This is identical to the RooParametricHist class already provided by Combine but take as input a TH2 instead of a TH1. The accompanying changes to accomodate this class are made in the Combine Tool code. Additionally, code has been added and modified in Combine Harvester (used for plotting) to plot the 2D distributions. User's not interested in development do not need to worry about the specifics of this since calls to RooParametricHist2D are all internal.

An example analysis is provided in Section **??** to show possible choices of D, x, and y and to illustrate the method graphically.

2 Prerequisites

2D Alphabet has only been tested in a CMSSW environment. The Higgs Analysis Combine Tool must be installed. Please follow the instructions below to checkout a 2D Alphabet-friendly version of Combine. These instructions are based off of the Combine documentation [1] for 102X setup. Please cross-check the instructions here with the official instructions. The only difference should be in the cloned repository and lack of branch change.

```
set SCRAM_ARCH=slc7_amd64_gcc700
cmsrel CMSSW_10_2_13
cd CMSSW_10_2_13/src
cmsenv
git clone https://github.com/lcorcodilos/HiggsAnalysis-CombinedLimit.git
    HiggsAnalysis/CombinedLimit
cd $CMSSW_BASE/src/HiggsAnalysis/CombinedLimit
scramv1 b clean; scramv1 b
cmsenv
```

If the command `combine --help` returns the help menu then you've successfully setup the environment.

Additionally, one needs the accompanying Combine Harvester setup in their `$CMSSW_BASE/src` which can be grabbed and compiled via

```
wget -O - https://raw.githubusercontent.com/lcorcodilos/CombineHarvester/master/
    CombineTools/scripts/sparse-checkout-ssh.sh | bash
```

3 Installation

Once a CMSSW environment with Combine is setup, one can install the 2D Alphabet software by simply cloning 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
```

Try to call RooParametricHist2D in interactive python if you're feeling uneasy and to ensure everything is working.

4 How To Run

The 2D Alphabet framework is designed to run by calling a one of a set of python scripts all with the prefix `run_`. The simplest and most important of these is `run_MLfit.py`.

4.1 Using run_MLfit.py

This script takes as input a list of JSON configuration files (detailed in Section **??**) via a command such as

```
python run_MLfit.py input_config1.json input_config2.json ... input_configN.json
```

where `input_config*.json` are the configurations for each Alphabet pair. Based on the input from the JSON files, the `run_MLfit.py` grabs all distributions, generates the RooFit objects from them, creates the pass and fail distributions of the QCD/non-resonant background estimated from data, passes these all to Combine, calls Combine, interprets the fit result, and plots the post-fit distributions. It also outputs additional plots and read-out for debugging or reference. `run_MLfit.py` creates the starting point for all other `run_` scripts because it

- Takes us from pre-fit (where we know little about the total background estimate) to post-fit where background estimate is meaningful,
- Creates a directory where all of the information of the run can be stored and accessed for later use.

Thus, `run_MLfit.py` should be run before the other scripts whenever attempting to fit a new Alphabet pair or a new set of pairs.

[1] `http://cms-analysis.github.io/HiggsAnalysis-CombinedLimit/`

Other possible command line options are provided below.

- `-q, --tag`: Assigns a tag for the run. This takes precendent over tags defined in configuration files and is a good way to make sure you don't overwrite an old run result by coming up with a new tag quickly.
- `--rMin/--rMax`: These are the minimum and maximum bounds of the signal strength (r) in the fit. They default to 0 and 5, respectively. It may be useful to loosen or tighten the bounds if the fit is failing near one of the boundaries.
- `--recycleAll`: Recycles everything generated in the previous run using the given tag (either by the –tag option or within the configuration file(s)). This means the construction of the workspace, fit guesses, and other steps previous to the fit are skipped and the previous run versions are loaded instead. Use this if you haven't changed your configuration file and would like to speed things up.
- `--skipFit`: Skips running the fit and goes directly to plotting.
- `--skipPlots`: Skips running the plots. Sometimes this is the part that takes the longest because a sampling method is used to estimate errors.

4.2 Using run_Limit.py

This script or is for calculating limits on the signal strength (r) of a given simulated signal. The script takes the background only fit result (`fit_b`) output from `run_MLfit.py`, morphs a copy of the pre-fit workspace to this result (thus bypassing having to perform a likelihood fit again), generates toys from the background estimate of the morphed pre-fit workspace to create pseudo-data, and fits these toys to derive expected limits for the simulated signal (blinding can be turned off so that the real data is also fit to get an observed limit on r).

Because the background only fit result is used, the signal simulation used in the configuration file for the `run_MLfit.py` does not matter. However, the signal for limit setting obviously matters and in all likelihood (no pun intended), the user will want to scan over several simulated signals to generate limits as a function of their samples (perhaps each one was generated at a different mass).

You do NOT have to write a configuration file for each simulated sample. At any point in the python call to `run_Limit.py`, one can provide a string swap specified by the syntax `oldString:newString` which will replace all instances of `oldString` in the configuration files provided with `newString`.

4.3 Using run_Impacts.py

4.4 Using run_Stats.py

4.5 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. You may delete these entries in your configuration file if you'd like. This section is a necessary supplement to those HELP keys.

Disclaimer: 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 it is the convention I'll use.

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 without lots of command line options. 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.** The six static sections are described below.

4.5.1 PROCESS

In this section, the user can define as many processes as they need to account for. This includes data, background simulation, and signal simulation. Please note two important things. (1) You should NOT define the background to be estimated via the transfer function (typically QCD). That background is naturally defined by the difference between data and the other backgrounds defined here. If you are attempting to estimate a background that is not QCD, you'll need to make this name change yourself. (2) Combine always requires that there be an observation (data), a background estimate, and a signal sample.

This means that your configuration file must contain at a minimum data (code 1) and signal (code 0) (it doesn't require a background because the QCD estimate will always exist).

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 (background simulation)

4.5.2 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: +1σ 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 +1σ 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 +1σ 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

[1] The codes classify the processes and are defined below

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 $+1\sigma$ pileup 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.r

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. It's entirely possible that a mistake has been made and a use case has not been account for correctly. If that has happened to you, please let me know so I can help and propagate any fix!

4.5.3 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 the 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 it *is* possible to rebin and reduce the ranges of the input axes. However, this is mainly for quick tests to remove a bin or reduce the number of bins. For permanant situations, my recommendation is to remake the input histograms to the desired binning and have the configuration file match (it's one less thing that can go wrong!)

The binning in each axis 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 (hopefully, this is also obvious). Additionally, the signal bounds only apply to the X axis and must exist within the X axis range defined in the configuration file. 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)
  - TITLE: title that you'd like to appear in plots with this 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)
  - TITLE: title that you'd like to appear in plots with this 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)

4.5.4 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. Expanding the number of functions available to the user is currently in development [2].

The user can convey the desired polynomial orders by separating the polynomial into XFORM and YFORM which are the shapes in the X and Y directions, respectively. For example, one could write (@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. Notice that the convention is that the numbering of the parameters in XFORM and YFORM must start at 1 - not 0.

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

When deciding on the nominal value and range for each parameter you should consider the following

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)
- ...

[2] please let me know if there's something you'd like me to include and I can do my best to implement it

4.5.5 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).

4.5.6 OPTIONS

This section is used to supply additional options to the fit that do not fit into any of the other five categories. A description of the possible options and the input they take are described below.

-

4.6 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`.