LGNDBDT

An Interpretable Boosted Decision Tree Model for Multi-Site Background Rejection for the LEGEND Collaboration.

Creators: Henry E. Nachman, with Aobo Li and Julieta Gruszko

For internal use by the LEGEND Collaboration

Latest Release: v1.0.1 - 17 July 2023

Last updated: July 18, 2023

Contents

1 Package Description

This package serves as a standalone code pipeline for use by the LEGEND Collaboration on LEGEND data to identify and reject multi-site events within Germanium detectors. This tool makes particular use of a Machine Learning (ML) method called a Boosted Decision Tree (BDT). BDTs operate by creating a series of decision trees, or networks of boolean decisions through which events are classified. It does this by training on pre-defined features extracted from the raw data (rather than the raw data itself). BDTS are a form of supervised ML methods, meaning they train on data that has been pre-labeled as well, in this case as either Single-Site, or Multi-Site. This manual will not go in depth on Boosted Decision Trees or Machine Learning, for more information please reference (1)

Because the model must take in pre-defined features, these features must first be chosen, then appropriately extracted from the raw event data. Over the past several decades of nuclear physics experiments, as well as previous experiments such as the MAJORANA DEMONSTRATOR, and Gerda, several waveform features have been identified and developed for use in waveform analysis. These parameters are largely known as Pulse Shape Discriminators (PSDs), and are the basis for the parameters used in this pipeline. For more information on each parameter, reference (2)

1.1 Motivation and Uses

In design, this package can be used to help identify and reject unwanted nuclear detections within the LEGEND experiment. In practice however, the accuracy and performance of the pipeline as a whole is not quite better than the extremely well-tuned, and carefully developed traditional methods used by the LEGEND collaboration. Perhaps with further development, this BDT model method for background rejection can become a more reliable tool for result analysis.

In the meantime, this tool is particularly useful as a fast analysis tool for detector characterization. Given that this package is entirely self-contained, it can be quickly and easily run on any machine. Additionally, as the pulse shape parameter extraction functions are primarily simplified versions of the highly-tuned methods traditionally used, using this code is likely faster and more efficient than passing raw data through the full LEGEND analysis pipeline. Thus, this package could serve as a quick (and dirty) analysis tool.

2 Acknowledgements

This work was largely completed in partial fulfillment of the requirements of Henry Nachman's Undergraduate Honors Thesis at the University of North Carolina at Chapel Hill under the tutelage of Dr. Julieta Gruszko, and Dr. Aobo Li.

This package makes use of other packages namely `LightGBM`[1], `SHAP`[3], and `PYGAMA`. `PYGAMA` is a publicly available software package created by the LEGEND Collaboration. More information about `PYGAMA` can be found at the following link https://pygama.readthedocs.io/en/stable/index.html.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under contract / award number DE-SC0022339

3 Installation

To install the package with the most recent updates, navigate to your python environment in terminal and run the code:

```
pip install git+https://github.com/henry-e-n/lgndbdt.git
```

To install a specific version of the package, add @ followed by the branch, commit hash, or tag name after the above command.

As the package is installing, it will automatically install any and all package dependencies needed to run the LGNDBDT code. After successful installation, you should receive a message that confirms the installed package version number.

Please note: this package is currently publicly available, but for security reasons does not include any data or information that could be considered proprietary by the LEGEND collaboration.

4 Package Components and Structure

A "Full Run" of the package consists of three main steps:

1. Waveform selection
2. Pulse Shape Discriminator extraction
3. Boosted Decision Tree model training

Each package component may be run independently, but ultimately must be run in order before model validation can continue.

For example, this package can be used as a relatively quick method for extracting PSDs from waveforms without necessarily training a BDT.

5 Input Data File Requirements

Likely the most tedious, and inflexible component of this package is the stringent requirements for input data. If the data is not stored exactly as defined in the following manner, simply put, the functions will be unable to find the required data, and may return errors.

I have tried to base the code on a standard way of file description and organization but despite this, users will inevitably encounter obstacles when running example code. As such, I encourage you to clone the package, and adapt it to your needs.

The data organization system is based on Morgan Clark's organization system for detector characterization data (which was also the basis for much of the code development).

5.1 Data Format

Each "run" should return two (2) files post initial processing.

One file, labeled "dsp" should be saved in LH5 file format. The "dsp" file should contain at a minimum datasets labeled $trapEmax$, and tp_0, with the estimated energy of the event, and start of the rise time respectively.

The other file, labeled "raw" (also saved in LH5 format), should contain at a minimum three datasets: $t0$ containing the starting time stamp of each waveform window (can be an array of zeros), dt the time step between data collection periods, and $values$ the physical recorded values that describe the waveform.

Luckily, so long as the datasets are labeled as described above, they can be nested within data groups on the LH5 file and can still be properly identified.

Ultimately, the five datasets needed and their size (where n stands for the total number of waveforms) are as follows:

- $trapEmax$, size: (n,)

- tp_0, size: (n,)
- $t0$, size: (n,)
- dt, size: (n,)
- *values*, size: (number of sample points, n)

5.2 File Organization

With these two data files create a folder structure as described below (note: <> indicate places to change with your own specifics).

```
<Data Folder>/dsp/<detector name>/<file name>.lh5
<Data Folder>/raw/<detector name>/<file name>.lh5
```

The file name should include 3 important pieces of information: run number, cycle number, and the detector name. This file name should be **identical** between the dsp and raw files, the distinction between the two is given by the folder name. For example, the file names are currently described as:

```
ornl-cyc<cycle#>-run<run#>-th228-{detector_name}
```

6 Example Run

6.1 Prologue

```
import numpy as np
import os, sys, json

pathToDSP = "<local path to dsp>/dsp/" # Defines local path to DSP folder
pathToRaw = "<local path to dsp>/raw/" # Defines local path to RAW folder

detector_name = "V00000A" # Defines the Detector Name

top_cyc_start     = 13 # Defines the first cycle number for the top source
top_cyc_end       = 89 # Defines the last cycle number (+1) for the top source
top_run_num       = 38  # Defines the run number for the top source

side_cyc_start     = 3 # Defines the first cycle number for the side source
side_cyc_end       = 13 # Defines the last cycle number (+1) for the side source
side_run_num       = 39 # Defines the run number for the side source

# Defines the path to which data files will be saved
savePath = "/<path to save folder>/" + detector_name

# Generates the list of filenames from which to pull data.
top_run_list = []
for run in range(top_cyc_start, top_cyc_end):
    top_run_list.append(f"ornl-cyc{run}-run{top_run_num}-th228-{detector_name}")
    # Change to correspond to appropriate file names
```

```
side_run_list = []
for run in range(side_cyc_start, side_cyc_end):
    side_run_list.append(f"ornl-cyc{run}-run{side_run_num}-th228-{detector_name}")
    # Change to correspond to appropriate file names


# Creates and writes to a json configuration file containing information on
# where the data is stores and which files to pull
jsonDict = {detector_name:{
                "top":{
                    "detector_name": detector_name,
                     "run_list": top_run_list,
                     "path_to_dsp": pathToDSP,
                     "path_to_raw": pathToRaw,
                     "file_save_path": savePath + "/top/",
                     "plot_save_path": savePath + "/top/plots/",
                     "source_loc": "top"},
                "side":{
                    "detector_name": detector_name,
                     "run_list": side_run_list,
                     "path_to_dsp": pathToDSP,
                     "path_to_raw": pathToRaw,
                     "file_save_path": savePath + "/side/",
                     "plot_save_path": savePath + "/side/plots/",
                     "source_loc": "side"}
                }
           }

jsonWrite = json.dumps(jsonDict, indent=4)
with open("paths.json", "w") as outfile:
    outfile.write(jsonWrite)
```

6.2 Waveform Selection and PSD extraction

```
import numpy as np
import os

# Necessary import statements
from lgndbdt.dev_env.run_files.raw_to_calibration      import calibrate_spectrum
from lgndbdt.dev_env.run_files.calibration_to_peakdata import extract_waveforms
from lgndbdt.dev_env.run_files.PSD_extraction          import psd_extraction
from lgndbdt.dev_env.utilities.get_files import get_save_paths, get_files
from lgndbdt.dev_env.utilities.h5_utils import search_file

f = open(f"{os.getcwd()}/paths.json")
data = json.load(f)
```

```
detector_list = (list(data.keys()))
for detector in detector_list: # loops over all specified detectors in JSON
    print(detector)
    detector_data = data[detector]
    source_loc_list = list(detector_data.keys())

    for source in source_loc_list: # loops over all sources in JSON
        print(source)
        data = detector_data[source]
        dsp_files, raw_files, file_save_path = get_files(data)

        ##############################################################################################

        # Defines the save paths and creates appropriate directories as needed
        file_save_path, plot_save_path = get_save_paths(detector, source)
        path_exists = os.path.exists(file_save_path)
        if not path_exists:
            os.makedirs(file_save_path)
            os.makedirs(plot_save_path)
            print(f"{detector} directory was created!")

        ##############################################################################################

        # Calibration Pass - Calibrates the spectra of each run number
        fit_exists = os.path.exists(f"{file_save_path}/fit_results_{source}.npy")
        # checks for existing fit
        if fit_exists:
            print(f"Using Pre-calibrated Fit found in {file_save_path}/fit_results_{source}.npy")
        else:
            cal_pars, [fitData, fit_pars]  = calibrate_spectrum(detector_name = detector,
                                                                source_location = source,
                                                                plots_bool=True)
            fitResults = np.array([fitData, fit_pars], dtype=object)
            np.save(f"{file_save_path}/fit_results_{source}.npy", fitResults, allow_pickle=True)
            np.save(f"{file_save_path}/cal_param_{source}.npy", cal_pars, allow_pickle=True)

        ##############################################################################################
        # WFD Extraction -
        # Extracts the waveforms from the appropriate peaks as defined in peak list
        fit_params  = np.load(f"{file_save_path}/fit_results_{source}.npy", allow_pickle=True)
        cal_params  = np.load(f"{file_save_path}/cal_param_{source}.npy", allow_pickle=True)

        peak_list = ["DEP", "SEP", "DEP_sideband", "SEP_sideband"]
        # peaks from which to pull waveforms

        psd_exists = os.path.exists(f"{file_save_path}/{detector}_PSDs_{peak_list[0]}_{source}.lh5")
        if psd_exists:
            print(f"File with PSDs already found --")
```

```
        else:
            # Save the current standard output
            original_stdout = sys.stdout
            # Open a null device as the new standard output
            dev_null = open(os.devnull, 'w')

            for target_peak in peak_list:
                print(target_peak)
                sys.stdout = dev_null # used to remove unnecessary print statements from function
                param_arr, param_keys  = extract_waveforms(detector_name = detector,
                                                           source_location = source,
                                                           calibration_parameters = cal_params,
                                                           fit_parameters = fit_params,
                                                           target_peak = target_peak)
                np.save(f"{file_save_path}/paramArr_{target_peak}.npy", param_arr)
                np.save(f"{file_save_path}/paramKeys_{target_peak}.npy", param_keys)
                # Extracts PSDs
                psd_extraction(param_arr, param_keys, detector, source, target_peak)
                # Restore the original standard output
                sys.stdout = original_stdout

# Close the null device
dev_null.close()
```

6.3 BDT training and validation

```
from lgndbdt.dev_env.run_files.BDT_train_validate import BDT_train_validate

detector_name = "V01387A" # Detector name
source_location = "mix"
# Defines whether to train on a mix of data or solely the top or side data

# Features for Training
train_features = ["/A_DAQE","/DCR", "/LQ80", "/TDRIFT", "/TDRIFT10", "/TDRIFT50"]
# Features for Distribution Matching
match_features    = ["/DCR", "/LQ80", "/TDRIFT", "/TDRIFT10", "/TDRIFT50"]
# Distribution Matching Steps
match_step     = [1e-5, 3, 5, 5, 5]

BDT_train_validate(detector_name, target_peak="SEP", source_location=source_location,
                   train_features=train_features, split_ratio = 0.3,
                   match_features=match_features, match_step=match_step)
```

7 Glossary of Functions

7.1 Primary Functions

```
    calibrate_spectrum(detector_name, source_location,
```

```
files_for_calibration=6,
verbose=False, plots_bool=False)
```

Description: Imports RAW data and runs a calibration program to convert the Analog to Digital Conversion (ADC) units to physical eV units. To help automate this process, the *pygama* python package, developed by members of the LEGEND analysis team, was used. The data undergoes a two-pass process to refine a good calibration fit. The first pass automatically identifies peaks in the energy spectrum and links them to a set of user-defined peaks for calibration. Five prominent peaks were chosen for calibration: ^{208}Tl FEP, DEP, and SEP as well as 1173.2 keV, and 1332.5 keV peaks resulting from ^{60}Co decay into ^{60}Ni (**?**). This calibration pass provides a good first guess at the calibration fit, but a more accurate fit of the spectrum and the individual peaks is necessary for event extraction. In the second calibration pass, each calibration peak is isolated from user-defined windows around the peak. Each peak is then fitted with a compound Gaussian and step function. From these fits, a highly accurate estimate of the mean energy and standard deviation of each peak is collected. Finally, the mean of each peak in ADC is fit to the accepted values in keV to yield a good calibration fit. This calibration process needs only be completed once for each run of data collection and thus can be accomplished without importing every event from the run. This saves a considerable amount of time and computational power as a good fit can be produced from a spectrum consisting of ~ 3 million events instead of the full $\sim 3.8 \times 10^9$ events collected. It is important to ensure that the calibration fit is accurate before proceeding with the code pipeline as a bad fit – often resulting from too little data or poorly defined peak windows – will result in improper extraction and labeling of events.

Parameter Name	Options	Description
detector_name	[<detector name>]	The detector name/serial number.
source_location	[top, side]	The location of the source during characterization.
files_for_calibration	[default=6]	Number of files to import for calibration, adjusting this number can help achieve a resolved calibration fit if the standard fit fails.
verbose	[default=FALSE]	Enables additional printing to console.
plots_bool	[default=TRUE]	Enables additional production and saving of plots.

Returns:
cal_pars: The parameters from the compound Gaussian fits of the calibration peaks.
[fit_Data, fit_pars]: The data and parameters from the linear calibration fit. Returns information needed to convert ADC for the run to eV.

```
extract_waveforms(detector_name, source_location,
                  calibration_parameters, fit_parameters,
                  target_peak, verbose=False)
```

Parameter Name	Options	Description
detector_name	[<detector name>]	The detector name/serial number.
source_location	[top, side]	The location of the source during characterization.
calibration_parameters	[cal_pars]	The parameters from the compound Gaussian fits of the calibration peaks – the return from calibrate_spectrum.
fit_parameters	[[fit_Data, fit_pars]]	The data and parameters from the linear calibration fit. Returns information needed to convert ADC for the run to eV – the return from calibrate_spectrum.
target_peak	[228ThDEP, 228ThSEP, 228ThDEP_sideband, 228ThSEP_sideband]	Specifies the window from which the waveforms will be extracted for this function.
verbose	[default=FALSE]	Enables additional printing to console.

Returns:
paramArr: An array of the necessary parameters trapEmax, t0, tp_0, dt, and values.
paramArrKeys: The names of the parameters included in paramArr, used to reference parameters later.

```
psd_extraction(paramArr, paramKeys,
               detector_name, source_location,
               target_peak)
```

Parameter Name	Options	Description
paramArr	[(parameters, # waveforms)]	An array of the necessary parameters trapEmax, t0, tp_0, dt, and values – the return of extract_waveforms.
paramKeys	[len(paramArr)]	The names of the parameters included in paramArr, used to reference parameters later – the return of extract_waveforms.
detector_name	[<detector name>]	The detector name/serial number.
source_location	[top, side]	The location of the source during characterization.
target_peak	[228ThDEP, 228ThSEP, 228ThDEP_sideband, 228ThSEP_sideband]	Specifies the window from which the waveforms will be extracted for this function.

Returns:
void:

```
BDT_train_validate(detector_name, target_peak, source_location,
                   train_features, match_features, match_step,
                   bdt_thresh = 0.55, avse_thresh = 969,
                   split_ratio = 0.3, validate="split",
                   augment = True, plots=True)
```

Parameter Name	Options	Description
detector_name	[<detector name>]	The detector name/serial number.
source_location	[top, side]	The location of the source during characterization.
target_peak	[228ThDEP, 228ThSEP, 228ThDEP_sideband, 228ThSEP_sideband]	Specifies the window from which the waveforms will be extracted for this function.
paramArr	[(parameters, # waveforms)]	An array of the necessary parameters trapEmax, t0, tp_0, dt, and values – the return of extract_waveforms.
paramKeys	[len(paramArr)]	The names of the parameters included in paramArr, used to reference parameters later – the return of extract_waveforms.

Returns:
void:

7.2 Secondary Functions

```
AvsE(values, dtimes, plots = [], numWF = 2500, currentEstimator = 100)
```

Function: Finds the maximum amplitude of the waveform by calculating the derivative of the wave via tensor arithmetic.
Arguments:
- ntimes array - [#Waveforms, #TimeCells]
- values - [#Waveforms, #TimeCells] - choose one waveform
- dtimes - [#Waveforms, #TimeCells]
Plots should either be:
- False - no plots to be printed
- E - energy array same length as times

```
find_slope_corr(values, valuesCorrected, dtimes)
```

Function: Finds the slope of the trailing tail of the Waveform after P0 correction
Parameters:
- values: Waveform ADC values
- valuesCorrected: P0 corrected waveform ADC values
- dtimes: time cell deltas
Returns:
- delta: Estimated Slope of tail

```
get_LQ80(ts, vals, trashPZ)
```

Function: Calculate the area about the turning edge of the waveform, aka LQ80
Parameters:
- ts: Waveform times array
- values: Waveform ADC values
- trashPZ: Array of waveform indeces to trash

```
get_PZ(vals, popt, numWave = 100)
```

Function: Performs P0 correction on waveforms
Parameters: - vals: Waveform ADC values [len(numWave), len(Cells)]
- popt: Boolean 0, runs minimization on P0 function to determine slope correction, uses input popt
- *numWave: number of waveforms
Returns: - wfIn: input waveforms in corresponding data structure
- wfCorr: Corrected waveforms

```
dp0Vis(popt, wfArray)
```

Function: Creates new waveforms with PZ correction.
Parameters:
- popt: The parameters of the PZ fit.
- wfArray: Array of raw waveforms to be corrected.
Returns:
- wfInAdj: RAW waveforms
- wfCorr: Corrected Waveforms
- trash: Array of waveforms to trash

```
get_TDRIFT_interpolate(times, values, startTime, dtimes)
```

Function: Determine the drift time values of a waveform.
Parameters:
- times: Times array.
- values: Waveform values:
- startTime: Start of the rise times:
- dtimes: dt of time samples.
Returns:
- tdrift
- tdrift50
- tdrift10

7.3 Tertiary and Miscellaneous Functions

```
us2cell(dtimes , ns = 1000)
```

Function: Takes in time (in ns) to be changed to cell width (based on dt)
Parameters: - dtimes: time cell deltas in ns (i think)
Returns:
- number of cells corresponding to time

```
box_average(values, index, side, dtimes, boxWidth = 1)
```

Function: Computes average ADC value of defined box.
Parameters:
- values: Waveform ADC values
- index: Index of evaluation box edge

- side: Which edge as given by index
- dtimes: time cell deltas
- *boxWidth = 1 defines the box width in microseconds
Returns
- avgBox: Average ADC value of box
- leftSide: Index of the left side of the box

```
find_percent(vals, buffer = 100, percent = 0.8)
```

Finds the index along waveform rise = given % of max

```
get_mid_index(ts, ind80, buffer = 100)
```

Finds the index of the middle of the flat tail used to split tail and compare rise integral to tail integral

```
dp0fx(popt, wfArray)
```

Function: Pole Zero function.
Parameters:
- popt: [tau1, tau2, f]
- tau1, tau2 - 2 exponential decay terms
- f - relation fraction
- wfArray: [len(Waveforms), ADC values len(cells)]
Returns:
- y_out: mean of standard deviation of fit waveforms

```
cTimes(wfParams, detName, savePath, number, save=True)
```

Creates an array of times for each waveform based on the t0 and dt given by paramArr.

References

[1] Ke, G. *et al.* LightGBM: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems* (2017). URL `https://papers.nips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html`.

[2] Nachman, H. E. Interpretable boosted decision tree model for multi-site background rejection in legend (2023).

[3] Lundberg, S. M. & Lee, S.-I. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems* (2017). URL `https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html`.