
actsncclass

Jul 31, 2023

Contents

1	Active Learning for Supernova Photometric Classification	1
1.1	Getting started	1
1.2	Analysis steps	3
1.3	Table of Contents	4
1.4	Indices and tables	42
Index		45

Active Learning for Supernova Photometric Classification

This tool allows you to reproduce the results presented in [Ishida et al., 2019](#). It was based on the original prototype developed during the [COIN Residence Program #4](#), which took place in Clermont Ferrand, France, August 2017.

The code has been updated to allow a friendly use and expansion.

1.1 Getting started

In order to setup a suitable working environment, clone this repository and make sure you have the necessary packages installed.

1.1.1 Dependencies

`actsnclass` was developed under `Python3`. The complete list of dependencies is given below:

- `Python` ≥ 3.7
- `matplotlib` $\geq 3.1.1$
- `numpy` $\geq 1.17.0$
- `pandas` $\geq 0.25.0$
- `setuptools` $\geq 41.0.1$
- `scipy` $\geq 1.3.0$
- `sklearn` $\geq 0.20.3$
- `seaborn` $\geq 0.9.0$

1.1.2 Installing

Clone this repository,

```
>>> git clone https://github.com/COINtoolbox/ActSNClass
```

We recommend the use of Anaconda environments to ensure the proper version of all dependencies are installed and do not interfere in your other applications. You can find instructions on how to install it [here](#).

If you wish to use this option, simply navigate to the directory of the repository and do:

```
>>> conda env create -f environment.yml
```

Once the environment is set up you can activate it:

```
>>> conda activate ActSNClass
```

If everything goes well you will see the name of the environment in the left most side of your command line.

You can now install *actsnclass* with:

```
(ActSNClass) >> python setup.py install
```

1.1.3 Setting up a working directory

In another location of your choosing, create the following directory structure:

```
work_dir
├── plots
└── results
```

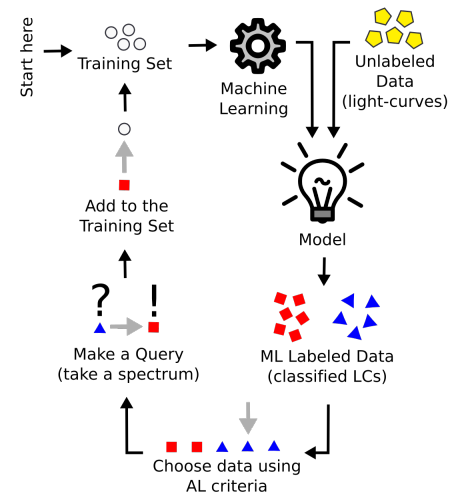
The outputs of *actsnclass* will be stored in these directories.

In order to set things properly, from the repository you just cloned, and move the data directory to your chosen working directory and unpack the data.

```
>>> mv -f actsnclass/data/ work_dir/
>>> cd work_dir/data
>>> tar -xzvf SIMGEN_PUBLIC_DES.tar.gz
```

This data was provided by Rick Kessler, after the publication of results from the [SuperNova Photometric Classification Challenge](#).

1.2 Analysis steps



The actsncclass pipeline is composed of 4 important steps:

1. Feature extraction
2. Classifier
3. Query Strategy
4. Metric evaluation

These are arranged in the adaptable learning process (figure to the right).

1.2.1 Using this package

Step 1 is considered pre-processing. The current code does the feature extraction using the [Bazin parametric function](#) for the complete training and test sample before any machine learning application is used.

Details of the tools available to evaluate different steps on feature extraction can be found in the [Feature extraction page](#).

Alternatively, you can also perform the full light curve fit for the entire sample from the command line:

```
>>> fit_dataset.py -dd <path_to_data_dir> -o <output_file>
```

Once the data has been processed you can apply the full Active Learning loop according to your needs. A detail description on how to use this tool is provided in the [Learning Loop page](#).

The command line option require a few more inputs than the feature extraction stage, but it is also available:

```
>>> run_loop.py -i <input features file> -b <batch size> -n <number of loops>
>>>                 -d <output metrics file> -q <output queried sample file>
>>>                 -s <learning strategy> -t <choice of initial training>
```

We also provide detail explanation on how to use this package to produce other stages of the pipeline like: [prepare the Canonical sample](#), [prepare data for time domain](#) and [produce plots](#).

We also provide detail descriptions on how to contribute with other modules in the [How to contribute](#) tab.

Enjoy!!

Acknowledgements

This work is heavily based on the first prototype developed during COIN Residence Program (CRP#4), held in Clermont Ferrand, France, 2017 and financially supported by [Universite Clermont Auvergne](#) and [La Region Auvergne-Rhone-Alpes](#). We thank Emmanuel Gangler for encouraging the realization of this event.

The [COsmostatistics INitiative \(COIN\)](#) receives financial support from [CNRS](#) as part of its MOMENTUM programme over the 2018-2020 period, under the project *Active Learning for Large Scale Sky Surveys*.

This work would not be possible without intensive consultation to online platforms and discussion forums. Although it is not possible to provide a complete list of the open source material consulted in the construction of this material, we recognize their importance and **deeply thank all those who contributes to open learning platforms**.

1.3 Table of Contents

1.3.1 Feature Extraction

The first stage in consists in transforming the raw data into a uniform data matrix which will subsequently be given as input to the learning algorithm.

The current implementation of `actsnclass` text-like data from the SuperNova Photometric Classification Challenge (SNPCC) which is described in [Kessler et al., 2010](#).

Processing 1 Light curve

The raw data looks like this:

```
SURVEY: DES
SNID: 848233
IAUC: UNKNOWN
PHOTOMETRY_VERSION: DES
SNTYPE: 22
FILTERS: griz
RA: 36.750000 deg
DECL: -4.500000 deg
MAGTYPE: LOG10
MAGREF: AB
FAKE: 2 (=> simulated LC with snlc_sim.exe)
MWEBV: 0.0283 MW E(B-V)
REDSHIFT_HELIO: 0.50369 +- 0.00500 (Helio, z_best)
REDSHIFT_FINAL: 0.50369 +- 0.00500 (CMB)
REDSHIFT_SPEC: 0.50369 +- 0.00500
REDSHIFT_STATUS: OK

HOST_GALAXY_GALID: 17173
HOST_GALAXY_PHOTO-Z: 0.4873 +- 0.0318

SIM_MODEL: NONIA 10 (name index)
SIM_NON1a: 30 (non1a index)
SIM_COMMENT: SN Type = II , MODEL = SDSS-017564
SIM_LIBID: 2
SIM_REDSHIFT: 0.5029
```

(continues on next page)

(continued from previous page)

```

SIM_HOSTLIB_TRUEZ: 0.5000 (actual Z of hostlib)
SIM_HOSTLIB_GALID: 17173
SIM_DLMU: 42.276020 mag [ -5*log10(10pc/dL) ]
SIM_RA: 36.750000 deg
SIM_DECL: -4.500000 deg
SIM_MWEBV: 0.0256 (MilkyWay E(B-V))
SIM_PEAKMAG: 22.48 22.87 22.70 22.82 (griz obs)
SIM_EXPOSURE: 1.0 1.0 1.0 1.0 (griz obs)
SIM_PEAKMJD: 56251.609375 days
SIM_SALT2x0: 1.229e-17
SIM_MAGDIM: 0.000
SIM_SEARCHEFF_MASK: 3 (bits 1,2=> found by software,humans)
SIM_SEARCHEFF: 1.0000 (spectro-search efficiency (ignores pipelines))
SIM_TRESTMIN: -38.24 days
SIM_TRESTMAX: 64.80 days
SIM_RISETIME_SHIFT: 0.0 days
SIM_FALLTIME_SHIFT: 0.0 days

SEARCH_PEAKMJD: 56250.734

# =====
# TERSE LIGHT CURVE OUTPUT:
#
NOBS: 108
NVAR: 9
VARLIST: MJD FLT FIELD FLUXCAL FLUXCALERR SNR MAG MAGERR SIM_MAG
OBS: 56194.145 g NULL 7.600e+00 4.680e+00 1.62 99.000 5.000 98.926
OBS: 56194.156 r NULL 3.875e+00 2.752e+00 1.41 99.000 5.000 98.953
OBS: 56194.172 i NULL 3.585e+00 4.628e+00 0.77 99.000 5.000 99.033
OBS: 56194.188 z NULL -2.203e+00 4.463e+00 -0.49 99.000 5.000 98.983
OBS: 56207.188 g NULL -7.008e+00 4.367e+00 -1.60 99.000 5.000 98.926
OBS: 56207.195 r NULL -1.189e+00 3.459e+00 -0.34 99.000 5.000 98.953
OBS: 56207.203 i NULL 8.799e+00 6.249e+00 1.41 99.000 5.000 99.033

```

You can load this data using:

```

1 >>> from actsncclass.fit_lightcurves import LightCurve
2
3 >>> path_to_lc = 'data/SIMGEN_PUBLIC_DES/DES_SN848233.DAT'
4
5 >>> lc = LightCurve() # create light curve instance
6 >>> lc.load_snpsc_lc(path_to_lc) # read data
7 >>> lc.photometry # check structure of photometry
8
9      mjd band flux fluxerr SNR
10 0 56194.145 g 7.600 4.680 1.62
11 1 56194.156 r 3.875 2.752 1.41
12 ... ... ...
13 106 56348.008 z 70.690 6.706 10.54
14 107 56348.996 g 26.000 5.581 4.66
[108 rows x 5 columns]

```

Once the data is loaded, you can fit each individual filter to the parametric function proposed by [Bazin et al., 2009](#) in one specific filter.

```

1 >>> rband_features = lc.fit_bazin('r')

```

(continues on next page)

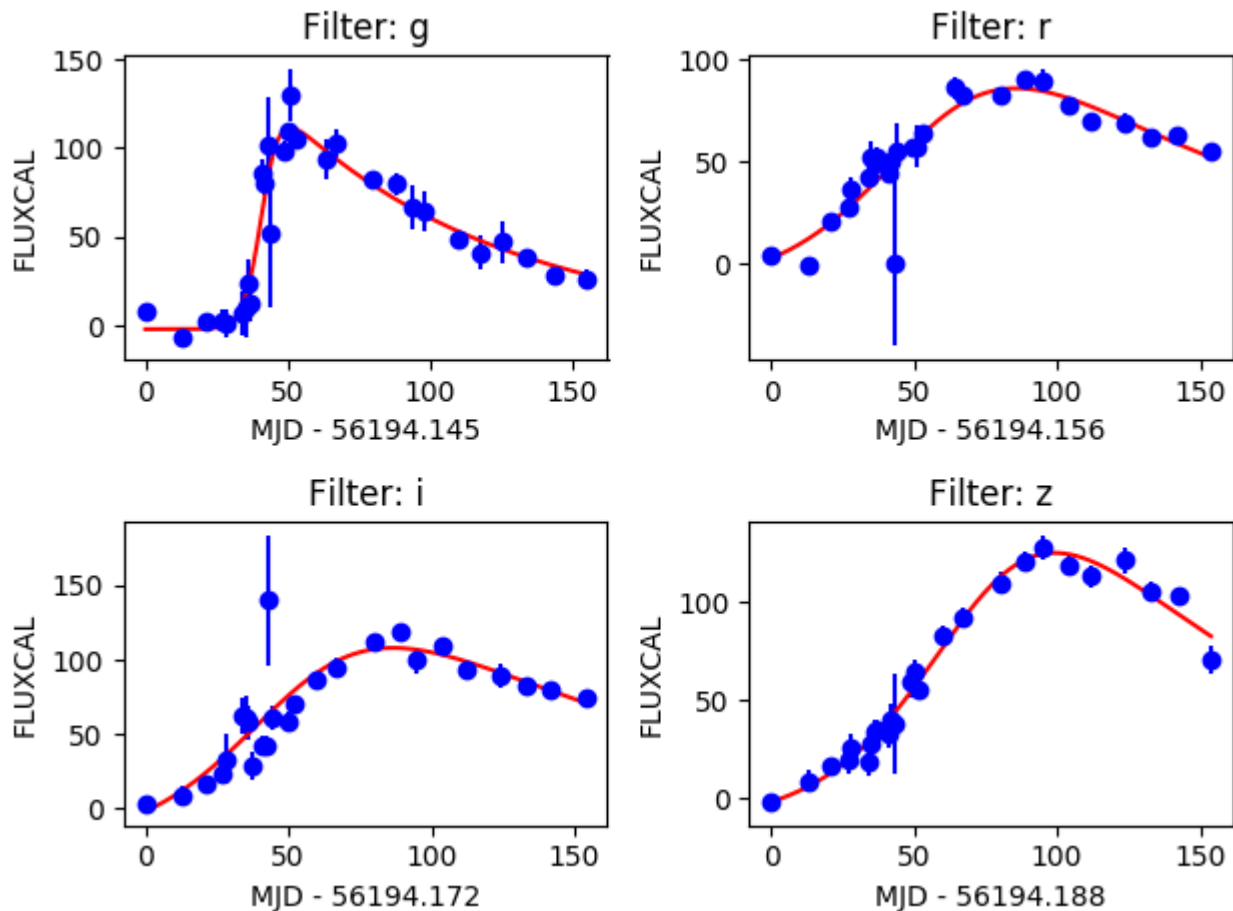
(continued from previous page)

```
2 >>> print(rband_features)
3 [159.25796385, -13.39398527, 55.16210333, 111.81204143, -20.13492354]
```

The designation for each parameter are stored in:

It is possible to perform the fit in all filters at once and visualize the result using:

```
1 >>> lc.fit_bazin_all() # perform Bazin fit in all filters
2 >>> lc.plot_bazin_fit(save=True, show=True,
3                       output_file='plots/SN' + str(lc.id) + '.png') # save to file
```



Processing all light curves in the data set

There are 2 way to perform the Bazin fits for the entire SNPCC data set. Using a python interpreter,

```
1 >>> from actsnclass import fit_snpcc_bazin
2
3 >>> path_to_data_dir = 'data/SIMGEN_PUBLIC_DES/' # raw data directory
4 >>> output_file = 'results/Bazin.dat' # output file
5 >>> fit_snpcc_bazin(path_to_data_dir=path_to_data_dir, features_file=output_file)
```

The above will produce a file called `Bazin.dat` in the `results` directory.

The same result can be achieved using the command line:

```
>> fit_dataset.py -dd <path_to_data_dir> -o <output_file>
```

1.3.2 Building the Canonical sample

According to the nomenclature used in [Ishida et al., 2019](#), the Canonical sample is a subset of the test sample chosen to hold the same characteristics of the training sample. It was used to mimic the effect of continuously adding elements to the training sample under the traditional strategy.

It was constructed using the following steps:

1. From the raw light curve files, build a metadata matrix containing: [snid, sample, sntype, z, g_pkmag, r_pkmag, i_pkmag, z_pkmag, g_SNR, r_SNR, i_SNR, z_SNR] where z corresponds to redshift, x_pkmag is the simulated peak magnitude and x_SNR denotes the mean SNR, both in filter x;
2. Separate original training and test set in 3 subsets according to SN type: [Ia, Ibc, II];
3. For each object in the training sample, find its nearest neighbor within objects of the test sample of the same SN type and considering the photometric parameter space built in step 1.

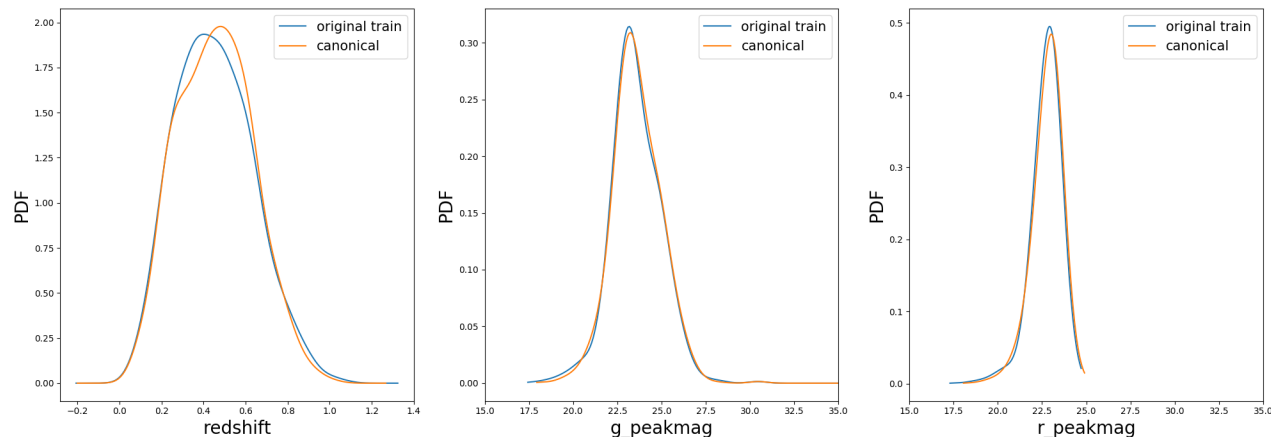
This will allow you to construct a Canonical sample holding the same characteristics and size of the original training sample but composed of different objects.

actsncclass allows you to perform this task using the `py:mod:actsncclass.build_snpcc_canonical` module:

```
1 >>> from snactclass import build_snpcc_canonical
2
3 >>> # define variables
4 >>> data_dir = 'data/SIMGEN_PUBLIC_DES/'
5 >>> output_sample_file = 'results/Bazin_SNPCC_canonical.dat'
6 >>> output_metadata_file = 'results/Bazin_metadata.dat'
7 >>> features_file = 'results/Bazin.dat'
8
9 >>> sample = build_snpcc_canonical(path_to_raw_data: data_dir, path_to_
  ↳ features=features_file,
10 >>>                                     output_canonical_file=output_sample_file,
11 >>>                                     output_info_file=output_metadata_file,
12 >>>                                     compute=True, save=True)
```

Once the samples is constructed you can compare the distribution in [z, g_pkmag, r_pkmag] with a plot:

```
1 >>> from actsncclass import plot_snpcc_train_canonical
2
3 >>> plot_snpcc_train_canonical(sample, output_plot_file='plots/compare_canonical_
  ↳ train.png')
```



In the command line, using the same parameters as in the code above, you can do all at once:

```
>>> build_canonical.py -c <if True compute metadata>
>>>     -d <path to raw data dir>
>>>     -f <input features file> -m <output file for metadata>
>>>     -o <output file for canonical sample> -p <comparison plot file>
>>>     -s <if True save metadata to file>
```

You can check that the file `results/Bazin_SNPCC_canonical.dat` is very similar to the original features file. The only difference is that now a few of the sample variables are set to queryable:

```
id redshift type code sample gA gB gt0 gtfall gtrise rA rB rt0 rtfall rtrise iA iB_
it0 itfall itrise zA zB zt0 ztfall ztrise
116537 0.5547 II 36 test 10.969309008063526 -2.505571025776927 33.36879338510094 89.
92091344407919 -1.4070121479476083 35.57261257957346 -0.97172916012906 47.
691951316763436 37.48483229249487 -7.146619117223875 41.16723042762342 0.
14005823764049471 47.983238664813264 39.02626334489017 -6.096248676680143 36.
82968789062783 -0.373638211418927 48.438610651533445 41.8848763303308 -7.
169183522127793
855370 0.5421 Ibc 23 test -5.514648646328689 3.370545820694393 6.890703579070343 127.
00223553079377 -0.04721760599586505 27.987087830949765 -0.4446376337515848 51.
06299616763716 13.46475077451422 -0.7802021055103384 42.50390337399486 1.
217778587283846 63.88727539461748 4.425762504064253 -2.826164280709543 57.
08358377564619 -0.9866672975549484 65.3976378960504 2.88096954432307 -2.
211749860376304
328118 0.3131 II 37 test 28.134338786167365 0.7147372066065217 45.830405214215425 15.
850284787778433 -0.0005766632993162762 29.225476277548275 -1.9734118280637896 45.
83230493446332 87.25700882127312 -0.00025821702716214264 24.95217257542528 -0.
3731568724509137 40.527255841246365 311.42509172517947 -3.099534332677601 46.
782672798921226 -0.05678675661798624 53.51739930097104 50.76716462668245 -4.
572685479766832
704481 0.4665 Ia 0 queryable -50.86850174812521 2.4148469184147547 16.05240678384717
3.5459318666713298 -0.4734666030325012 74.65602268994473 -3.763616485144308 48.
208444944828855 24.3318092539982 -4.452287612782472 83.29745588526693 -7.
371877954771961 50.92270461365078 38.76468635410394 -10.931632426569717 73.
35112115534632 -1.2509966370291774 40.053959252846106 44.453394158157614 -0.
18674652754319326
43679 0.5756 II 33 test 28.24271470397688 -3.438072722932048 23.521675700587007 32.
4401288159836 -0.2295765027048151 -37.86668398190429 6.8580060036559365 22.
252525376185087 2.3940753934318044 -1.611409074593934 -20.420915833911547 9.
2659565057976 7.0218302478113035 23.713442135755557 -0.027609543521757457 14.
76124690750807 -5.175821286895905 32.58560788340983 115.86494837233313 -0.
2587648450330448
```

(continues on next page)

(continued from previous page)

```

172648 0.7592 II 31 test -5.942021205498 2.5808480681448858 72.24216865162195 83.
↪43696533883242 -0.04052830859563986 17.05919635984848 1.005755998955811 18.
↪148318002391164 33.16119808959254 -0.12803412647871454 12.153694246253906 -1.
↪2962293252577974 17.493068792921502 89.98548146319197 -0.14950758787782462 13.
↪355316206445695 -2.4143982246591293 23.84002028961246 118.87985827861259 -1.
↪4858837093947788
762146 0.7245 Ibc 22 test 10.734014319410377 -0.696725251384634 92.36623187978644 0.
↪5112285753252996 -0.4900012400030447 12.968161724599275 -0.94670057528261 55.
↪7516252880299 25.59410571631452 -1.971658945324412 19.03779421586546 -2.
↪228264147418322 57.66412361316971 35.09222360219662 -3.325944814741228 22.
↪877393161444374 0.3070939958786501 57.9675613727551 49.63155574417528 -1.
↪8832424871891025

```

This means that you can use the `actsncclass.learn_loop` module in combination with a RandomSampling strategy but reading data from the canonical sample. In this way, at each iteration the code will select a random object from the test sample but a query will only be made if the selected object belongs to the canonical sample.

In the command line, this looks like:

```

>>> run_loop.py -i results/Bazin_SNPCC_canonical.dat -b <batch size> -n <number of_
↪loops>
>>> -d <output metrics file> -q <output queried sample file>
>>> -s RandomSampling -t <choice of initial training>

```

1.3.3 Prepare data for time domain

In order to mimic the realistic situation where only a limited number of observed epochs is available at each day, it is necessary to prepare our simulate data resemble this scenario. In `actsncclass` this is done in 5 steps:

1. Determine minimum and maximum MJD for the entire SNPCC sample;
2. For each day of the survey, run through the entire data sample and select only the observed epochs which were obtained prior to it;
3. Perform the feature extraction process considering only the photometric points which survived item 2.
4. Check if in the MJD in question the object is available for querying.
5. Join all information in a standard features file.

You can perform the entire analysis for one day of the survey using the `actsncclass.time_domain` module:

```

1 >>> from actsncclass.time_domain import SNPCCPhotometry
2
3 >>> path_to_data = 'data/SIMGEN_PUBLIC_DES/'
4 >>> output_dir = 'results/time_domain/'
5 >>> day = 20
6
7 >>> data = SNPCCPhotometry()
8 >>> data.create_daily_file(output_dir=output_dir, day=day)
9 >>> data.build_one_epoch(raw_data_dir=path_to_data, day_of_survey=day,
10                        time_domain_dir=output_dir)

```

Alternatively you can use the command line to prepare a sequence of days in one batch:

```

>>> build_time_domain.py -d 20 21 22 23 -p <path to raw data dir> -o <path to output_
↪time domain dir>

```

1.3.4 Active Learning loop

Details on running 1 loop

Once the data has been pre-processed, analysis steps 2-4 can be performed directly using the DataBase object.

For start, we can load the feature information:

```
1 >>> from actsnclass import DataBase
2
3 >>> path_to_features_file = 'results/Bazin.dat'
4
5 >>> data = DataBase()
6 >>> data.load_features(path_to_features_file, method='Bazin')
7 Loaded 21284 samples!
```

Notice that this data has some pre-determine separation between training and test sample:

```
1 >>> data.metadata['sample'].unique()
2 array(['test', 'train'], dtype=object)
```

You can choose to start your first iteration of the active learning loop from the original training sample flagged in the file OR from scratch. As this is our first example, let's do the simple thing and start from the original training sample. The code below build the respective samples and performs the classification:

```
1 >>> data.build_samples(initial_training='original', nclass=2)
2 Training set size: 1093
3 Test set size: 20191
4
5 >>> data.classify(method='RandomForest')
6 >>> data.classprob # check classification probabilities
7 array([[0.461, 0.539],
8        [0.346, 0.654],
9        ...,
10       [0.398, 0.602],
11       [0.396, 0.604]])
```

Hint: If you wish to start from scratch, just set the *initial_training=N* where *N* is the number of objects in you want in the initial training. The code will then randomly select *N* objects from the entire sample as the initial training sample. It will also impose that at least half of them are SNe Ias.

For a binary classification, the output from the classifier for each object (line) is presented as a pair of floats, the first column corresponding to the probability of the given object being a Ia and the second column its complement.

Given the output from the classifier we can calculate the metric(s) of choice:

```
1 >>> data.evaluate_classification(metric_label='snpsc')
2 >>> print(data.metrics_list_names) # check metric header
3 ['acc', 'eff', 'pur', 'fom']
4
5 >>> print(data.metrics_list_values) # check metric values
6 [0.5975434599574068, 0.9024767801857585,
7 0.34684684684684686, 0.13572404702012383]
```

and save results for this one loop to file:

```

1 >>> path_to_features_file = 'results/Bazin.dat'
2 >>> metrics_file = 'results/metrics.dat'
3 >>> queried_sample_file = 'results/queried_sample.dat'
4
5 >>> data.save_metrics(loop=0, output_metrics_file=metrics_file)
6 >>> data.save_queried_sample(loop=0, queried_sample_file=query_file,
7 >>>                           full_sample=False)

```

You should now have in your results directory a metrics.dat file which looks like this:

```

day accuracy efficiency purity fom query_id
0 0.4560942994403447 0.5545490350531705 0.23933367329593744 0.05263972502898026 81661

```

Running a number of iterations in sequence

We provide a function where all the above steps can be done in sequence for a number of iterations. In interactive mode, you must define the required variables and use the `actsnclass.learn_loop` function:

```

1 >>> from actsnclass.learn_loop import learn_loop
2
3 >>> nloops = 1000                                # number of iterations
4 >>> method = 'Bazin'                             # only option in v1.0
5 >>> ml = 'RandomForest'                          # only option in v1.0
6 >>> strategy = 'RandomSampling'                   # learning strategy
7 >>> input_file = 'results/Bazin.dat'               # input features file
8 >>> metric = 'results/metrics.dat'                 # output metrics file
9 >>> queried = 'results/queried.dat'                # output query file
10 >>> train = 'original'                            # initial training
11 >>> batch = 1                                     # size of batch
12
13 >>> learn_loop(nloops=nloops, features_method=method, classifier=ml,
14 >>>             strategy=strategy, path_to_features=input_file, output_metrics_
15 >>>             ↪file=metrics,
16 >>>             output_queried_file=queried, training=train, batch=batch)

```

Alternatively you can also run everything from the command line:

```

>>> run_loop.py -i <input features file> -b <batch size> -n <number of loops>
>>>               -d <output metrics file> -q <output queried sample file>
>>>               -s <learning strategy> -t <choice of initial training>

```

The queryable sample

In the example shown above, when reading the data from the features file there was only 2 possibilities for the *sample* variable:

```

1 >>> data.metadata['sample'].unique()
2 array(['test', 'train'], dtype=object)

```

This corresponds to an unrealistic scenario where we are able to obtain spectra for any object at any time.

Hint: If you wish to restrict the sample available for querying, just change the *sample* variable to *queryable* for the objects available for querying. Whenever this keyword is encountered in a file of extracted features, the code

automatically restricts the query selection to the objects flagged as *queryable*.

1.3.5 Active Learning loop in time domain

Considering that you have previously prepared the time domain data, you can run the active learning loop in its current form either by using the `actsnclass.time_domain_loop` or by using the command line interface:

```
>>> run_time_domain.py -d <first day of survey> <last day of survey>
>>>     -m <output metrics file> -q <output queried file> -f <features directory>
>>>     -s <learning strategy> -t <choice of initial training>
```

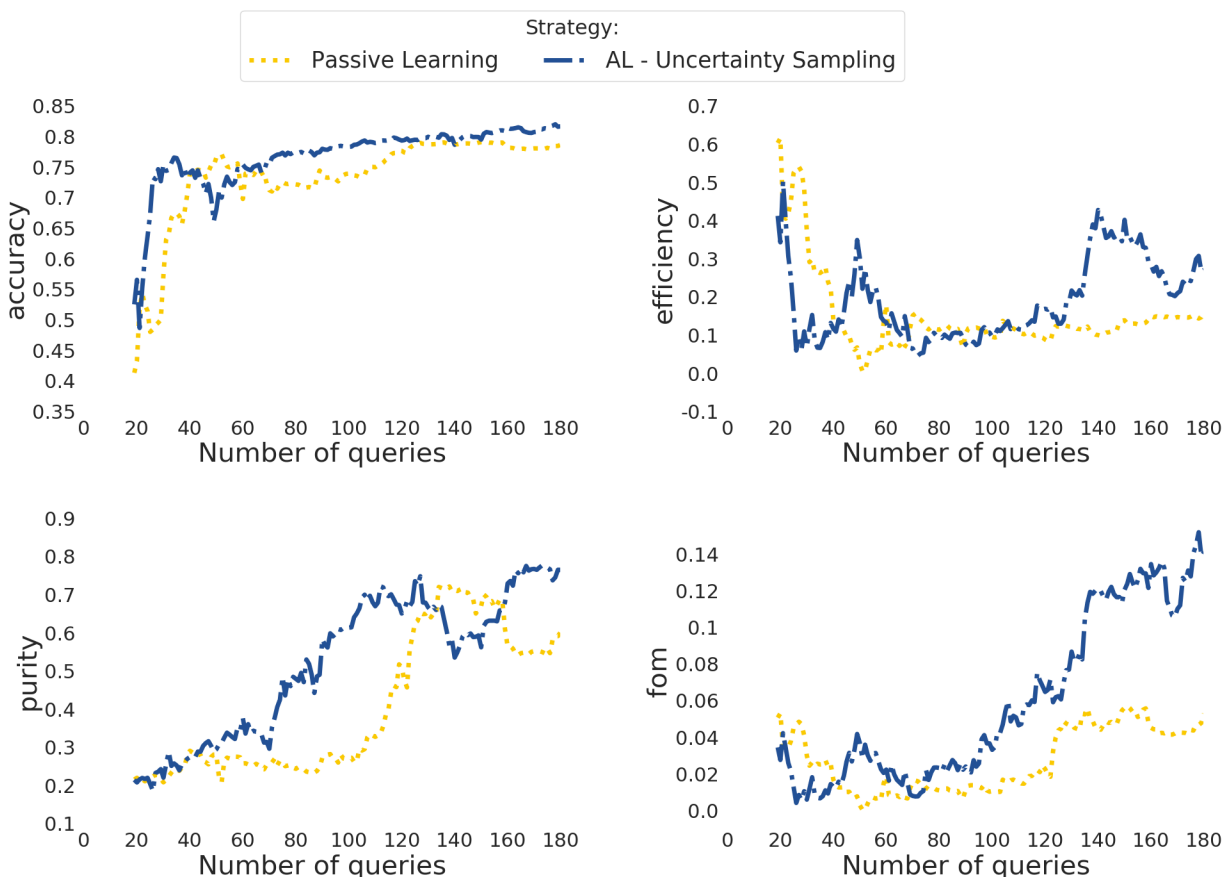
Make sure you check the full documentation of the module to understand which variables are required depending on the case you wish to run.

For example, to run with SNPCC data, the largest survey interval you can run is between 20 and 182 days, the corresponding option will be `-d 20 182`.

In the example above, if you choose to start from the original training sample, `-t original` you must also input the path to the file containing the full light curve analysis - so the full initial training can be read. This option corresponds to `-t original -fl <path to full lc features>`.

More details can be found in the corresponding [docstring](#).

Once you ran one or more options, you can use the `actsnclass.plot_results` module, as described in the [produce plots](#) page. The result will be something like the plot below (accounting for variations due to initial training).



Warning: At this point there is no *Canonical sample* option implemented for the time domain module.

1.3.6 Plotting

Once you have the metrics results for a set of learning strategies you can plot the behaviour the evolution of the metrics:

- Accuracy: fraction of correct classifications;
- Efficiency: fraction of total SN Ia correctly classified;
- Purity: fraction of correct Ia classifications;
- Figure of merit: efficiency x purity with a penalty factor of 3 for false positives (contamination).

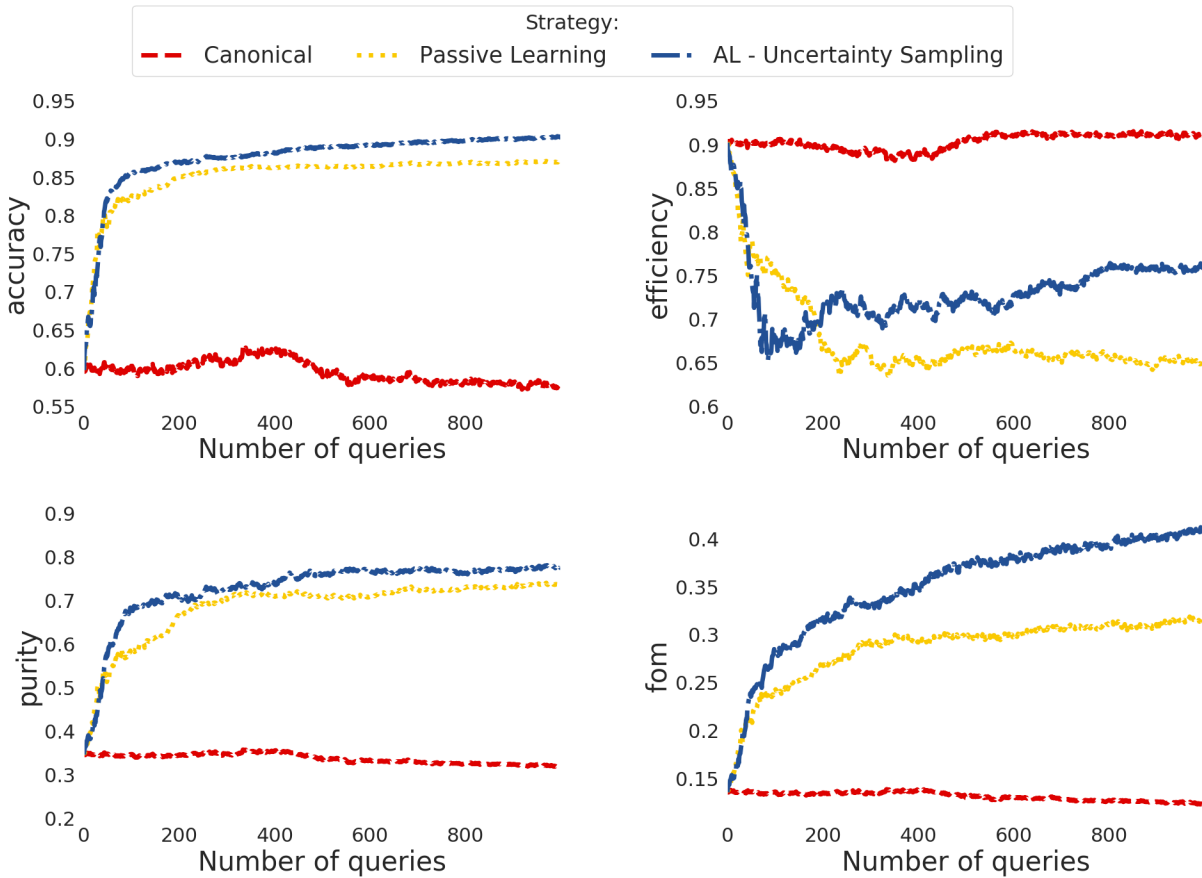
The class *Canvas* <<https://actsncclass.readthedocs.io/en/latest/api/actsncclass.Canvas.html#actsncclass.Canvas>>_ enables you do to it using:

```

1  >>> from actsncclass.plot_results import Canvas
2
3  >>> # define parameters
4  >>> path_to_files = ['results/metrics_canonical.dat',
5  >>>                  'results/metrics_random.dat',
6  >>>                  'results/metrics_unc.dat']
7  >>> strategies_list = ['Canonical', 'RandomSampling', 'UncSampling']
8  >>> output_plot = 'plots/metrics.png'
9
10 >>> #Initiate the Canvas object, read and plot the results for
11 >>> # each metric and strategy.
12 >>> cv = Canvas()
13 >>> cv.load_metrics(path_to_files=path_to_files,
14 >>>                  strategies_list=strategies_list)
15 >>> cv.set_plot_dimensions()
16 >>> cv.plot_metrics(output_plot_file=output_plot,
17 >>>                  strategies_list=strategies_list)

```

This will generate:



Alternatively, you can use it directly from the command line.

For example, the result above could also be obtained doing:

```
>>> make_metrics_plots.py -m <path to canonical metrics> <path to rand sampling_
    metrics> <path to unc sampling metrics>
>>> -o <path to output plot file> -s Canonical RandomSampling UncSampling
```

OBS: the color palette for this project was chosen to honor the work of [Piet Mondrian](#).

1.3.7 How to contribute

Below you will find general guidance on how to prepare your piece of code to be integrated to the actsnclass environment.

Add a new data set

The main challenge of adding a new data set is to build the infrastructure necessary to handle the new data.

The function below show how the basic structure required to deal with 1 light curve:

```
1 >>> import pandas as pd
2
3 >>> def load_one_lightcurve(path_to_data, *args):
4 >>>     """Load 1 light curve at a time.
```

(continues on next page)

(continued from previous page)

```

5  >>>
6  >>>     Parameters
7  >>>     -----
8  >>>     path_to_data: str
9  >>>         Complete path to data file.
10 >>>     ...
11 >>>     ...
12 >>>
13 >>>     Returns
14 >>>     -----
15 >>>     pd.DataFrame
16 >>>     """
17 >>>
18 >>>     #####
19 >>>     # Do something #####
20 >>>     #####
21 >>>
22 >>>     # structure of light curve
23 >>>     lc = {}
24 >>>     lc['dataset_name'] = XXXX           # name of the data set
25 >>>     lc['filters'] = [X, Y, Z]         # list of filters
26 >>>     lc['id'] = XXX                    # identification number
27 >>>     lc['redshift'] = X                 # redshift (optional, important for_
    ↪building canonical)
28 >>>     lc['sample'] = XXXXX              # train, test or queryable (none is_
    ↪mandatory)
29 >>>     lc['sntype'] = X                  # Ia or non-Ia
30 >>>     lc['photometry'] = pd.DataFrame() # min keys: MJD, filter, FLUX, FLUXERR
31 >>>                                     # bonus: MAG, MAGERR, SNR
32 >>>     return lc

```

Feel free to also provide other keywords which might be important to handle your data. Given a function like this we should be capable of incorporating it into the pipeline.

Please refer to the `actsnclass.fit_lightcurves` module for a closer look at this part of the code.

Add a new feature extraction method

Currently `actsnclass` only deals with Bazin features. The snippet below show an example of friendly code for a new feature extraction method.

```

1  >>> def new_feature_extraction_method(time, flux, *args):
2  >>>     """Extract features from light curve.
3  >>>
4  >>>     Parameters
5  >>>     -----
6  >>>     time: 1D - np.array
7  >>>         Time of observation.
8  >>>     flux: 1D - np.array of floats
9  >>>         Measured flux.
10 >>>     ...
11 >>>     ...
12 >>>
13 >>>     Returns
14 >>>     -----
15 >>>     set of features

```

(continues on next page)

(continued from previous page)

```

16 >>> """
17 >>>
18 >>> #####
19 >>> ### Do something #####
20 >>> #####
21 >>>
22 >>> return features

```

You can check the current feature extraction tools for the Bazin parametrization at `actsnclass.bazin` module.

Add a new classifier

A new classifier should be wrap in a function such as:

```

1 >>> def new_classifier(train_features, train_labels, test_features, *args):
2 >>>     """Random Forest classifier.
3 >>>
4 >>>     Parameters
5 >>>     -----
6 >>>     train_features: np.array
7 >>>         Training sample features.
8 >>>     train_labels: np.array
9 >>>         Training sample classes.
10 >>>     test_features: np.array
11 >>>         Test sample features.
12 >>>     ...
13 >>>     ...
14 >>>
15 >>>     Returns
16 >>>     -----
17 >>>     predictions: np.array
18 >>>         Predicted classes - 1 class per object.
19 >>>     probabilities: np.array
20 >>>         Classification probability for all objects, [pIa, pnon-Ia].
21 >>>     """
22 >>>
23 >>> #####
24 >>> ##### Do something #####
25 >>> #####
26 >>>
27 >>> return predictions, probabilities

```

The only classifier implemented at this point is a Random Forest and can be found at the `actsnclass.classifiers` module.

Important: Remember that in order to be effective in the active learning frame work a classifier should not be heavy on the required computational resources and must be sensitive to small changes in the training sample. Otherwise the evolution will be difficult to tackle.

Add a new query strategy

A query strategy is a protocol which evaluates the current state of the machine learning model and makes an informed decision about which objects should be included in the training sample.

This is very general, and the function can receive as input any information regarding the physical properties of the test and/or target samples and current classification results.

A minimum structure for such function would be:

```

1  >>> def new_query_strategy(class_prob, test_ids, queryable_ids, batch, *args):
2  >>>     """New query strategy.
3  >>>
4  >>>     Parameters
5  >>>     -----
6  >>>     class_prob: np.array
7  >>>         Classification probability. One value per class per object.
8  >>>     test_ids: np.array
9  >>>         Set of ids for objects in the test sample.
10 >>>     queryable_ids: np.array
11 >>>         Set of ids for objects available for querying.
12 >>>     batch: int
13 >>>         Number of objects to be chosen in each batch query.
14 >>>     ...
15 >>>     ...
16 >>>
17 >>>     Returns
18 >>>     -----
19 >>>     query_indx: list
20 >>>         List of indexes identifying the objects from the test sample
21 >>>         to be queried in decreasing order of importance.
22 >>>     """
23 >>>
24 >>>     #####
25 >>>     ##### Do something #####
26 >>>     #####
27 >>>
28 >>>     return list of indexes of size batch

```

The current available strategies are Passive Learning (or Random Sampling) and Uncertainty Sampling. Both can be scrutinized at the :py:mod:actsnclass.'query_strategies' module.

Add a new diagnostic metric

Beyond the criteria for choosing an object to be queried one could also think about the possibility to test different metrics to evaluate the performance of the classifier at each learning loop.

A new diagnostic metrics can then be provided in the form:

```

1  >>> def new_metric(label_pred: list, label_true: list, ia_flag, *args):
2  >>>     """Calculate efficiency.
3  >>>
4  >>>     Parameters
5  >>>     -----
6  >>>     label_pred: list
7  >>>         Predicted labels
8  >>>     label_true: list
9  >>>         True labels
10 >>>     ia_flag: number, symbol
11 >>>         Flag used to identify Ia objects.
12 >>>     ...
13 >>>     ...

```

(continues on next page)

(continued from previous page)

```

14 >>>
15 >>>     Returns
16 >>>     -----
17 >>>     a number or set of numbers
18 >>>     Tells us how good the fit was.
19 >>>     """
20 >>>
21 >>>     #####
22 >>>     ##### Do something ! #####
23 >>>     #####
24 >>>
25 >>>     return a number or set of numbers

```

The currently implemented diagnostic metrics are those used in the SNPCC (Kessler et al., 2009) and can be found at the `actsncclass.metrics` module.

1.3.8 Reference / API

Pre-processing

Light curve analysis

Performing feature extraction for 1 light curve

<code>LightCurve()</code>	Light Curve object, holding meta and photometric data.
<code>LightCurve.load_snpsc_lc(path_to_data)</code>	Reads one LC from SNPCC data.
<code>LightCurve.fit_bazin(band)</code>	Extract Bazin features for one filter.
<code>LightCurve.fit_bazin_all()</code>	Perform Bazin fit for all filters independently and concatenate results.
<code>LightCurve.plot_bazin_fit([save, show, ...])</code>	Plot data and Bazin fitted function.

actsncclass.LightCurve

class `actsncclass.LightCurve`

Light Curve object, holding meta and photometric data.

Variables

- **bazin_features_names** (*list*) – List of names of the Bazin function parameters.
- **bazin_features** (*list*) – List with the 5 best-fit Bazin parameters in all filters. Concatenated from blue to red.
- **dataset_name** (*str*) – Name of the survey or data set being analyzed.
- **filters** (*list*) – List of broad band filters.
- **id** (*int*) – SN identification number
- **photometry** (*pd.DataFrame*) – Photometry information. Keys → [mjd, band, flux, fluxerr, SNR, MAG, MAGERR].
- **redshift** (*float*) – Redshift
- **sample** (*str*) – Original sample to which this light curve is assigned

- **sim_peakmag** (*np.array*) – Simulated peak magnitude in each filter
- **sncode** (*int*) – Number identifying the SN model used in the simulation
- **sntype** (*str*) – General classification, possibilities are: Ia, II or Ibc

check_queryable (*mjd: float, r_lim: float*)

Check if this light can be queried in a given day.

load_snpcc_lc (*path_to_data: str*)

Reads header and photometric information for 1 light curve

fit_bazin (*band: str*) → list

Calculates best-fit parameters from the Bazin function in 1 filter

fit_bazin_all ()

Calculates best-fit parameters from the Bazin func for all filters

plot_bazin_fit (*save: bool, show: bool, output_file: str*)

Plot photometric points and Bazin fitted curve

Examples

```
>>> from actsncclass import LightCurve
```

define path to light curve file

```
>>> path_to_lc = 'data/SIMGEN_PUBLIC_DES/DES_SN431546.DAT'
```

```
>>> lc = LightCurve()                                # create light curve instance
>>> lc.load_snpcc_lc(path_to_lc)                     # read data
>>> lc.photometry                                     # display photometry
```

	mjd	band	flux	fluxerr	SNR
0	56207.188	g	9.6560	4.369	2.21
1	56207.195	r	6.3370	3.461	1.83
...
96	56336.043	r	14.4300	3.098	4.66
97	56336.055	i	18.9500	5.029	3.77

[98 rows x 5 columns]

```
>>> lc.fit_bazin_all()                               # perform Bazin fit in all filters
>>> lc.bazin_features                                # display Bazin parameters
```

[62.0677260096896,	-7.959383808822104,	47.37511467606875,	37.4919069623379,
...
206.65806244385922,	-4.777010246622081]		

plot light curve fit

```
>>> lc.plot_bazin_fit(output_file=str(lc.id) + '.png')
```

for fitting the entire sample...

```
>>> path_to_data_dir = 'data/SIMGEN_PUBLIC_DES/'      # raw data directory
>>> output_file = 'results/Bazin.dat'                 # output file
>>> fit_snpcc_bazin(path_to_data_dir=path_to_data_dir, features_file=output_file)
```

a file with all Bazin fits for this data set was produced

__init__()
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>check_queryable(mjd, r_lim)</code>	Check if this light can be queried in a given day.
<code>fit_bazin(band)</code>	Extract Bazin features for one filter.
<code>fit_bazin_all()</code>	Perform Bazin fit for all filters independently and concatenate results.
<code>load_snpsc_lc(path_to_data)</code>	Reads one LC from SNPCC data.
<code>plot_bazin_fit([save, show, output_file])</code>	Plot data and Bazin fitted function.

actsncclass.LightCurve.load_snpsc_lc

`LightCurve.load_snpsc_lc(path_to_data: str)`

Reads one LC from SNPCC data.

Populates the attributes: `dataset_name`, `id`, `sample`, `redshift`, `sncode`, `sntype`, `photometry` and `sim_peakmag`.

Parameters `path_to_data` (*str*) – Path to text file with data from a single SN.

actsncclass.LightCurve.fit_bazin

`LightCurve.fit_bazin(band: str)`

Extract Bazin features for one filter.

Parameters `band` (*str*) – Choice of broad band filter

Returns `bazin_param` – Best fit parameters for the Bazin function: [a, b, t0, tfall, trise]

Return type list

actsncclass.LightCurve.fit_bazin_all

`LightCurve.fit_bazin_all()`

Perform Bazin fit for all filters independently and concatenate results.

Populates the attributes: `bazin_features`.

actsncclass.LightCurve.plot_bazin_fit

`LightCurve.plot_bazin_fit(save=True, show=False, output_file='')`

Plot data and Bazin fitted function.

Parameters

- `save` (*bool (optional)*) – Save figure to file. Default is True.
- `show` (*bool (optional)*) – Display plot in window. Default is False.
- `output_file` (*str*) – Name of file to store the plot.

Fitting an entire data set

<code>fit_snpcc_bazin(path_to_data_dir, features_file)</code>	Fit Bazin functions to all filters in training and test samples.
---	--

actsnclclass.fit_snpcc_bazin

`actsnclclass.fit_snpcc_bazin(path_to_data_dir: str, features_file: str)`

Fit Bazin functions to all filters in training and test samples.

Parameters

- **path_to_data_dir** (*str*) – Path to directory containing the set of individual files, one for each light curve.
- **features_file** (*str*) – Path to output file where results should be stored.

Basic light curve analysis tools

<code>bazin.bazin(time, a, b, t0, tfall, trise)</code>	Parametric light curve function proposed by Bazin et al., 2009.
<code>bazin.errfunc(params, time, flux)</code>	Absolute difference between theoretical and measured flux.
<code>bazin.fit_scipy(time, flux)</code>	Find best-fit parameters using <code>scipy.least_squares</code> .

actsnclclass.bazin.bazin

`actsnclclass.bazin.bazin(time, a, b, t0, tfall, trise)`

Parametric light curve function proposed by Bazin et al., 2009.

Parameters

- **time** (*np.array*) – exploratory variable (time of observation)
- **a** (*float*) – Normalization parameter
- **b** (*float*) – Shift parameter
- **t0** (*float*) – Time of maximum
- **tfall** (*float*) – Characteristic decline time
- **trise** (*float*) – Characteristic raise time

Returns response variable (flux)

Return type array_like

actsnclclass.bazin.errfunc

`actsnclclass.bazin.errfunc(params, time, flux)`

Absolute difference between theoretical and measured flux.

Parameters

- **params** (*list of float*) – light curve parameters: (a, b, t0, tfall, trise)
- **time** (*array_like*) – exploratory variable (time of observation)

- **flux** (*array_like*) – response variable (measured flux)

Returns **diff** – absolute difference between theoretical and observed flux

Return type float

actsnclass.bazin.fit_scipy

actsnclass.bazin.**fit_scipy** (*time, flux*)

Find best-fit parameters using scipy.least_squares.

Parameters

- **time** (*array_like*) – exploratory variable (time of observation)
- **flux** (*array_like*) – response variable (measured flux)

Returns **output** – best fit parameter values

Return type list of float

Canonical sample

The Canonical object for holding the entire sample.

<code>Canonical()</code>	Canonical sample object.
<code>Canonical.snpcc_get_canonical_info(...[, ...])</code>	Load SNPCC metadata data required to characterize objects.
<code>Canonical.snpcc_identify_samples()</code>	Identify training and test sample.
<code>Canonical.find_neighbors()</code>	Identify 1 nearest neighbor for each object in training.

actsnclass.Canonical

class actsnclass.**Canonical**

Canonical sample object.

Variables

- **canonical_ids** (*list*) – List of ids for objects in the canonical sample.
- **canonical_sample** (*list*) – Complete data matrix for the canonical sample.
- **meta_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for all objects in the original data set.
- **test_ia_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN Ias in the test sample.
- **test_ia_id** (*np.array*) – Set of ids for all SN Ia in the test sample.
- **test_ibc_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN Ibcs in the test sample.
- **test_ibc_id** (*np.array*) – Set of ids for all SN Ibc in the test sample.
- **test_ii_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN IIs in the test sample.
- **test_ii_id** (*np.array*) – Set of ids for all SN II in the test sample.

- **train_ia_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN Ias in the train sample.
- **train_ia_id** (*np.array*) – Set of ids for all SN Ia in the train sample.
- **train_ibc_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN Ibcs in the train sample.
- **train_ibc_id** (*np.array*) – Set of ids for all SN Ibc in the train sample.
- **train_ii_data** (*pd.DataFrame*) – Metadata on sim peakmag and SNR for SN IIs in the train sample.
- **train_ii_id** (*np.array*) – Set of ids for all SN II in the train sample.

snpsc_get_canonical_info (*path_to_rawdata_dir: str, canonical_output_file: str, compute: bool, save: bool, canonical_input_file: str*)
Load SNPCC metada data required to characterize objects.

snpsc_identify_samples ()
Identify training and test sample.

find_neighbors ()
Identify 1 nearest neighbor for each object in training.

__init__ ()
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>find_neighbors()</code>	Identify 1 nearest neighbor for each object in training.
<code>snpsc_get_canonical_info(...[, compute, ...])</code>	Load SNPCC metada data required to characterize objects.
<code>snpsc_identify_samples()</code>	Identify training and test sample.

actsncclass.Canonical.snpsc_get_canonical_info

Canonical.snpsc_get_canonical_info (*path_to_rawdata_dir: str, canonical_output_file: str, compute=True, save=True, canonical_input_file=""*)
Load SNPCC metada data required to characterize objects.

Populates attribute: data.

Parameters

- **path_to_rawdata_dir** (*str*) – Complete path to directory holding raw data files.
- **canonical_output_file** (*str*) – Complete path to output canonical sample file.
- **compute** (*bool (optional)*) – Compute required metada from raw data files. Default is True.
- **save** (*bool (optional)*) – Save metadata to file. Default is True.
- **canonical_input_file** (*str (optional)*) – Path to input file if required metadata was previously calculated. If name is give, 'compute' must be False.

actsnclass.Canonical.snpcc_identify_samples

`Canonical.snpcc_identify_samples()`

Identify training and test sample.

Populates attributes: `train_ia_data`, `train_ia_id`, `train_ibc_data`, `train_ibc_id`, `train_ii_data`, `train_ibc_id`, `test_ia_data`, `test_ia_id`, `test_ibc_data`, `test_ibc_id`, `test_ii_data` and `test_ii_id`.

actsnclass.Canonical.find_neighbors

`Canonical.find_neighbors()`

Identify 1 nearest neighbor for each object in training.

Populates attribute: `canonical_ids`.

Functions to populate the Canonical object

<code>build_snpcc_canonical(path_to_raw_data, ...)</code>	Build canonical sample for SNPCC data.
<code>plot_snpcc_train_canonical(sample[, ...])</code>	Plot comparison between training and canonical samples.

actsnclass.build_snpcc_canonical

`actsnclass.build_snpcc_canonical` (*path_to_raw_data*: str, *path_to_features*: str, *output_canonical_file*: str, *output_info_file*=", *compute*=True, *save*=True, *input_info_file*=", *features_method*='Bazin')

Build canonical sample for SNPCC data.

Parameters

- **path_to_raw_data** (*str*) – Complete path to raw data directory.
- **path_to_features** (*str*) – Complete path to Bazin features files.
- **output_canonical_file** (*str*) – Complete path to output canonical sample file.
- **output_info_file** (*str*) – Complete path to output metadata file for canonical sample. This includes SNR and simulated peak magnitude for each filter.
- **compute** (*bool (optional)*) – If True, compute metadata information on SNR and sim peak mag. If False, read info from file. Default is True.
- **save** (*bool (optional)*) – Save simulation metadata information to file. Default is True.
- **input_info_file** (*str (optional)*) – Complete path to sim metadata file. This must be provided if `save == False`.
- **features_method** (*str (optional)*) – Method for feature extraction. Only 'Bazin' is implemented.

Returns `actsnclass.Canonical` – Updated canonical object with the attribute 'canonical_sample'.

Return type obj

actsncclass.plot_snpcc_train_canonical

`actsncclass.plot_snpcc_train_canonical` (*sample: actsncclass.build_snpcc_canonical.Canonical*,
output_plot_file=False)

Plot comparison between training and canonical samples.

Parameters

- **sample** (*actsncclass.Canonical*) – Canonical object holding infor for canonical sample
- **output_plot_file** (*str (optional)*) – Complete path to output plot. If not provided, plot is displayed on screen only.

Build time domain data base

<code>SNPCCPhotometry()</code>	Handles photometric information for entire SNPCC data.
<code>SNPCCPhotometry.get_lim_mjds(raw_data_dir)</code>	Get minimum and maximum MJD for complete sample.
<code>SNPCCPhotometry.create_daily_file(...[, header])</code>	Create one file for a given day of the survey.
<code>SNPCCPhotometry.build_one_epoch(...[, ...])</code>	Fit bazin for all objects with enough points in a given day.

actsncclass.SNPCCPhotometry

class `actsncclass.SNPCCPhotometry`

Handles photometric information for entire SNPCC data.

This class only works for Bazin feature extraction method.

Variables

- **bazin_header** (*str*) – Reader to be added to features files for each day.
- **max_epoch** (*float*) – Maximum MJD for the entire data set.
- **min_epoch** (*float*) – Minimum MJD for the entire data set.
- **rmag_lim** (*float*) – Maximum r-band magnitude allowing a query.

get_lim_mjds (*raw_data_dir*)

Get minimum and maximum MJD for complete sample.

create_daily_file (*raw_data_dir: str, day: int, output_dir: str, header: str*)

Create one file for a given day of the survey. Only populates the file with header. It will erase existing files!

build_one_epoch (*raw_data_dir: str, day_of_survey: int, time_domain_dir: str, feature_method: str, dataset: str*)

Selects objects with observed points until given MJD, performs feature extraction and evaluate if query is possible. Save results to file.

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>build_one_epoch(raw_data_dir, day_of_survey, ...)</code>	Fit bazin for all objects with enough points in a given day.
<code>create_daily_file(output_dir, day[, header])</code>	Create one file for a given day of the survey.
<code>get_lim_mjds(raw_data_dir)</code>	Get minimum and maximum MJD for complete sample.

actsnclass.SNPCCPhotometry.get_lim_mjds

`SNPCCPhotometry.get_lim_mjds(raw_data_dir)`

Get minimum and maximum MJD for complete sample.

This function is not necessary if you are working with SNPCC data. The values are hard coded in the class.

Parameters `raw_data_dir` (*str*) – Complete path to raw data directory.

Returns `limits` – List of extreme MJDs for entire sample: [min_MJD, max_MJD].

Return type list

actsnclass.SNPCCPhotometry.create_daily_file

`SNPCCPhotometry.create_daily_file(output_dir: str, day: int, header='Bazin')`

Create one file for a given day of the survey.

The file contains only header for the features file.

Parameters

- **output_dir** (*str*) – Complete path to raw data directory.
- **day** (*int*) – Day passed since the beginning of the survey.
- **header** (*str optional*) – List of elements to be added to the header. Separate by 1 space. Default option uses header for Bazin features file.

actsnclass.SNPCCPhotometry.build_one_epoch

`SNPCCPhotometry.build_one_epoch(raw_data_dir: str, day_of_survey: int, time_domain_dir: str, feature_method='Bazin', dataset='SNPCC')`

Fit bazin for all objects with enough points in a given day.

Generate 1 file containing best-fit Bazin parameters for a given day of the survey.

Parameters

- **raw_data_dir** (*str*) – Complete path to raw data directory
- **day_of_survey** (*int*) – Day since the beginning of survey.
- **time_domain_dir** (*str*) – Output directory to store time domain files.
- **feature_method** (*str optional*) – Feature extraction method. Only possibility is 'Bazin'.
- **dataset** (*str optional*) – Name of the data set. Only possibility is 'SNPCC'.

DataBase

Object upon which the learning process is performed

<code>DataBase()</code>	DataBase object, upon which the active learning loop is performed.
<code>DataBase.load_bazin_features(path_to_bazin_file)</code>	Load Bazin features from file.
<code>DataBase.load_features(path_to_file[, ...])</code>	Load features according to the chosen feature extraction method.
<code>DataBase.build_samples(initial_training[, ...])</code>	Separate train and test samples.
<code>DataBase.classify([method, screen, n_est, ...])</code>	Apply a machine learning classifier.
<code>DataBase.evaluate_classification([...])</code>	Evaluate results from classification.
<code>DataBase.make_query([strategy, batch, seed, ...])</code>	Identify new object to be added to the training sample.
<code>DataBase.update_samples(query_indx, loop[, ...])</code>	Add the queried obj(s) to training and remove them from test.
<code>DataBase.save_metrics(loop, ...[, batch])</code>	Save current metrics to file.
<code>DataBase.save_queried_sample(...[, ...])</code>	Save queried sample to file.

actsnnclass.DataBase

class actsnnclass.DataBase

DataBase object, upon which the active learning loop is performed.

Variables

- **classprob** (`np.array()`) – Classification probability for all objects, [pla, pnon-la].
- **data** (`pd.DataFrame`) – Complete information read from features files.
- **features** (`pd.DataFrame()`) – Feature matrix to be used in classification (no meta-data).
- **features_names** (`list`) – Header for attribute *features*.
- **metadata** (`pd.DataFrame`) – Features matrix which will not be used in classification.
- **metadata_names** (`list`) – Header for metadata.
- **metrics_list_names** (`list`) – Values for metric elements.
- **predicted_class** (`np.array()`) – Predicted classes - results from ML classifier.
- **queried_sample** (`np.array()`) – Complete information of queried objects.
- **queryable_ids** (`np.array()`) – Flag for objects available to be queried.
- **test_features** (`pd.DataFrame`) – Features matrix for the test sample.
- **test_metadata** (`pd.DataFrame()`) – Metadata for the test sample
- **test_labels** (`np.array()`) – True classification for the test sample.
- **train_features** (`pd.DataFrame()`) – Features matrix for the train sample.
- **train_metadata** (`pd.DataFrame()`) – Metadata for the training sample.
- **train_labels** (`np.array()`) – Classes for the training sample.

load_bazin_features (`path_to_bazin_file: str`)

Load Bazin features from file

load_features (*path_to_file: str, method: str*)
Load features according to the chosen feature extraction method.

build_samples (*initial_training: str or int, nclass: int*)
Separate train and test samples.

classify (*method: str*)
Apply a machine learning classifier.

evaluate_classification (*metric_label: str*)
Evaluate results from classification.

make_query (*strategy: str, batch: int*) → list
Identify new object to be added to the training sample.

update_samples (*query_indx: list*)
Add the queried obj(s) to training and remove them from test.

save_metrics (*loop: int, output_metrics_file: str*)
Save current metrics to file.

save_queried_sample (*queried_sample_file: str, loop: int, full_sample: str*)
Save queried sample to file.

Examples

```
>>> from actsncclass import DataBase
```

Define the necessary paths

```
>>> path_to_bazin_file = 'results/Bazin.dat'
>>> metrics_file = 'results/metrics.dat'
>>> query_file = 'results/query_file.dat'
```

Initiate the DataBase object and load the data. `>>> data = DataBase()` `>>> data.load_features(path_to_bazin_file, method='Bazin')`

Separate training and test samples and classify

```
>>> data.build_samples(initial_training='original', nclass=2)
>>> data.classify(method='RandomForest')
>>> print(data.classprob)           # check predicted probabilities
[[0.461 0.539]
[0.346print(data.metrics_list_names)           # check metric header
['acc', 'eff', 'pur', 'fom']]
```

```
>>> print(data.metrics_list_values)           # check metric values
[0.5975434599574068, 0.9024767801857585,
0.34684684684684686, 0.13572404702012383] 0.654]
...
[0.398 0.602]
[0.396 0.604]]
```

Calculate classification metrics

```
>>> data.evaluate_classification(metric_label='snpc')
>>>
```

Make query, choose object and update samples


```
>>> indx = data.make_query(strategy='UncSampling', batch=1)
>>> data.update_samples(indx)
```

Save results to file

```
>>> data.save_metrics(loop=0, output_metrics_file=metrics_file)
>>> data.save_queried_sample(loop=0, queried_sample_file=query_file,
>>>                           full_sample=False)
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>build_samples(initial_training[, nclass, ...])</code>	Separate train and test samples.
<code>classify([method, screen, n_est, seed, ...])</code>	Apply a machine learning classifier.
<code>evaluate_classification([metric_label, screen])</code>	Evaluate results from classification.
<code>load_bazin_features(path_to_bazin_file[, screen])</code>	Load Bazin features from file.
<code>load_features(path_to_file[, method, screen])</code>	Load features according to the chosen feature extraction method.
<code>make_query([strategy, batch, seed, screen])</code>	Identify new object to be added to the training sample.
<code>save_metrics(loop, output_metrics_file, epoch)</code>	Save current metrics to file.
<code>save_queried_sample(queried_sample_file, loop)</code>	Save queried sample to file.
<code>update_samples(query_indx, loop[, epoch, screen])</code>	Add the queried obj(s) to training and remove them from test.

actsnclass.DataBase.load_bazin_features

`DataBase.load_bazin_features(path_to_bazin_file: str, screen=False)`

Load Bazin features from file.

Populate properties: data, features, feature_list, header and header_list.

Parameters

- **path_to_bazin_file** (*str*) – Complete path to Bazin features file.
- **screen** (*bool (optional)*) – If True, print on screen number of light curves processed. Default is False.

actsnclass.DataBase.load_features

`DataBase.load_features(path_to_file: str, method='Bazin', screen=False)`

Load features according to the chosen feature extraction method.

Populates properties: data, features, feature_list, header and header_list.

Parameters

- **path_to_file** (*str*) – Complete path to features file.
- **method** (*str (optional)*) – Feature extraction method. The current implementation only accepts `method=='Bazin'`
- **screen** (*bool (optional)*) – If True, print on screen number of light curves processed. Default is False.

bactsnclaff.Datadbaff.builld_sampless

Datadbaff.**builld_sampless** (*initial_training: str, nclaff=2, screen=False, queryable=False*)

Separate train and test sampless.

Populate properties: `train_features`, `train_header`, `test_features`, `test_header`, `queryable_ids` (if flag available), `train_labels` and `test_labels`.

Parameters

- **initial_training** (*str or int*) – Choice of initial training sample. If 'original': begin from the train sample flagged in the file If int: choose the required number of sampless at random, ensuring that at least half are SN Ia.
- **nclaff** (*int (optional)*) – Number of claffess to consider in the claffification Currently only `nclaff == 2` is implemented.
- **screen** (*bool (optional)*) – If True display the dimensions of training and test sampless.
- **queryable** (*bool (optional)*) – If True build also queryable sample for time domain analysis. Default is False.

bactsnclaff.Datadbaff.claffify

Datadbaff.**claffify** (*method='RandomForest', screen=False, n_est=1000, seed=42, max_depth=None, n_jobs=1*)

Apply a machine learning claffifier.

Populate properties: `predicted_class` and `claff_prob`

Parameters

- **method** (*str (optional)*) – Chosen claffifier. The current implementation on accepts *RandomForest*.
- **n_est** (*int (optional)*) – Number of trees. Default is 1000.
- **screen** (*bool (optional)*) – If True, print debug comments on screen. Default is False.
- **seed** (*int (optional)*) – Random seed. Default is 42.
- **max_depth** (*None or int (optional)*) – The maximum depth of the tree. Default is None.
- **n_jobs** (*int (optional)*) – Number of cores used to train the model. Default is 1.

bactsnclaff.Datadbaff.evaluate_claffification

Datadbaff.**evaluate_claffification** (*metric_label='snpsc', screen=False*)

Evaluate results from claffification.

Populate properties: `metric_list_names` and `metrics_list_values`.

Parameters **metric_label** (*str*) – Choice of metric. Currently only *snpc* is accepted.

actsnclass.DataBase.make_query

`DataBase.make_query (strategy='UncSampling', batch=1, seed=42, screen=False) → list`

Identify new object to be added to the training sample.

Parameters

- **seed** (*int (optional)*) – Random seed. Default is 42.
- **strategy** (*str (optional)*) – Strategy used to choose the most informative object. Current implementation accepts 'UncSampling' and 'RandomSampling'. Default is *UncSampling*.
- **batch** (*int (optional)*) – Number of objects to be chosen in each batch query. Default is 1.
- **screen** (*bool (optional)*) – If true, display on screen information about the displacement in order and classification probability due to constraints on queryable sample.

Returns **query_indx** – List of indexes identifying the objects to be queried in decreasing order of importance. If `strategy=='RandomSampling'` the order is irrelevant.

Return type list

actsnclass.DataBase.update_samples

`DataBase.update_samples (query_indx: list, loop: int, epoch=0, screen=False)`

Add the queried obj(s) to training and remove them from test.

Update properties: `train_headers`, `train_features`, `train_labels`, `test_labels`, `test_headers` and `test_features`.

Parameters

- **query_indx** (*list*) – List of indexes identifying objects to be moved.
- **loop** (*int*) – Store number of loop when this query was made.
- **epoch** (*int (optional)*) – Initial epoch since survey started. Default is 0.
- **screen** (*bool (optional)*) – Print debug auxiliary information on screen. Default is False.

actsnclass.DataBase.save_metrics

`DataBase.save_metrics (loop: int, output_metrics_file: str, epoch: int, batch=1)`

Save current metrics to file.

If `loop == 0` the 'output_metrics_file' will be created or overwritten. Otherwise results will be added to an existing 'output_metrics file'.

Parameters

- **loop** (*int*) – Number of learning loops finished at this stage.
- **output_metrics_file** (*str*) – Full path to file to store metrics results.
- **batch** (*int*) – Number of queries in each loop.
- **epoch** (*int*) – Days since the beginning of the survey.

actsnclass.DataBase.save_queried_sample

`DataBase.save_queried_sample` (*queried_sample_file*: str, *loop*: int, *full_sample*=False, *batch*=1)
Save queried sample to file.

Parameters

- **queried_sample_file** (str) – Complete path to output file.
- **loop** (int) – Number of learning loops finished at this stage.
- **full_sample** (bool (optional)) – If true, write down a complete queried sample stored in property 'queried_sample'. Otherwise append 1 line per loop to 'queried_sample_file'. Default is False.

Classifiers

<code>random_forest</code> (train_features, train_labels, ...)	Random Forest classifier.
--	---------------------------

actsnclass.random_forest

`actsnclass.random_forest` (*train_features*: numpy.array, *train_labels*: numpy.array, *test_features*: numpy.array, *nest*=1000, *seed*=42, *max_depth*=None, *n_jobs*=1)
Random Forest classifier.

Parameters

- **train_features** (np.array) – Training sample features.
- **train_labels** (np.array) – Training sample classes.
- **test_features** (np.array) – Test sample features.
- **nest** (int (optional)) – Number of estimators (trees) in the forest. Default is 1000.
- **seed** (float (optional)) – Seed for random number generator. Default is 42.
- **max_depth** (None or int (optional)) – The maximum depth of the tree. Default is None.
- **n_jobs** (int (optional)) – Number of cores used to train the model. Default is 1.

Returns

- **predictions** (np.array) – Predicted classes.
- **prob** (np.array) – Classification probability for all objects, [pnon-Ia, pIa].

Query strategies

<code>random_sampling</code> (test_ids, queryable_ids[, ...])	Randomly choose an object from the test sample.
<code>uncertainty_sampling</code> (class_prob, test_ids, ...)	Search for the sample with highest uncertainty in predicted class.

actsnclass.random_sampling

`actsnclass.random_sampling` (*test_ids*: *numpy.array*, *queryable_ids*: *numpy.array*, *batch*=1, *seed*=42, *screen*=False) → list

Randomly choose an object from the test sample.

Parameters

- **test_ids** (*np.array*) – Set of ids for objects in the test sample.
- **queryable_ids** (*np.array*) – Set of ids for objects available for querying.
- **batch** (*int (optional)*) – Number of objects to be chosen in each batch query. Default is 1.
- **screen** (*bool (optional)*) – If True display on screen the ids of queried objects. Default is False.
- **seed** (*int (optional)*) – Seed for random number generator. Default is 42.

Returns **query_idx** – List of indexes identifying the objects from the test sample to be queried.

Return type list

actsnclass.uncertainty_sampling

`actsnclass.uncertainty_sampling` (*class_prob*: *numpy.array*, *test_ids*: *numpy.array*, *queryable_ids*: *numpy.array*, *batch*=1, *screen*=False) → list

Search for the sample with highest uncertainty in predicted class.

Parameters

- **class_prob** (*np.array*) – Classification probability. One value per class per object.
- **test_ids** (*np.array*) – Set of ids for objects in the test sample.
- **queryable_ids** (*np.array*) – Set of ids for objects available for querying.
- **batch** (*int (optional)*) – Number of objects to be chosen in each batch query. Default is 1.
- **dump** (*bool (optional)*) – If True display on screen the shift in index and the difference in estimated probabilities of being Ia caused by constraints on the sample available for querying.

Returns **query_idx** – List of indexes identifying the objects from the test sample to be queried in decreasing order of importance.

Return type list

Metrics*Individual metrics*

<code>accuracy</code> (label_pred, label_true)	Calculate accuracy.
<code>efficiency</code> (label_pred, label_true[, ia_flag])	Calculate efficiency.
<code>purity</code> (label_pred, label_true[, ia_flag])	Calculate purity.
<code>fom</code> (label_pred, label_true[, ia_flag, penalty])	Calculate figure of merit.

actsnclass.accuracy

`actsnclass.accuracy` (*label_pred: list, label_true: list*)

Calculate accuracy.

Parameters

- **label_pred** (*list*) – predicted labels
- **label_true** (*list*) – true labels

Returns **Accuracy** – Global fraction of correct classifications.

Return type float

actsnclass.entropy

`actsnclass.entropy` (*label_pred: list, label_true: list, ia_flag=1*)

Calculate entropy.

Parameters

- **label_pred** (*list*) – Predicted labels
- **label_true** (*list*) – True labels
- **ia_flag** (*int (optional)*) – Flag used to identify Ia objects. Default is 1.

Returns **entropy** – Fraction of correctly classified SN Ia.

Return type float

actsnclass.purity

`actsnclass.purity` (*label_pred: list, label_true: list, ia_flag=1*)

Calculate purity.

Parameters

- **label_pred** (*list*) – Predicted labels
- **label_true** (*list*) – True labels
- **ia_flag** (*int (optional)*) – Flag used to identify Ia objects. Default is 1.

Returns **Purity** – Fraction of true SN Ia in the final classified Ia sample.

Return type float

actsnclass.fom

`actsnclass.fom` (*label_pred: list, label_true: list, ia_flag=1, penalty=3.0*)

Calculate figure of merit.

Parameters

- **label_pred** (*list*) – Predicted labels
- **label_true** (*list*) – True labels
- **ia_flag** (*bool (optional)*) – Flag used to identify Ia objects. Default is 1.

- **penalty** (*float*) – Weight given for non-Ias wrongly classified.

Returns figure of merit – Efficiency x pseudo-purity (purity with a penalty for false positives)

Return type float

Metrics aggregated by category or use

<code>get_snpcc_metric(label_pred, label_true[, ...])</code>	Calculate the metric parameters used in the SNPCC.
--	--

actsnclass.get_snpcc_metric

`actsnclass.get_snpcc_metric` (*label_pred: list, label_true: list, ia_flag=1, wpenalty=3*)

Calculate the metric parameters used in the SNPCC.

Parameters

- **label_pred** (*list*) – Predicted labels
- **label_true** (*list*) – True labels
- **ia_flag** (*bool (optional)*) – Flag used to identify Ia objects. Default is 1.
- **wpenalty** (*float*) – Weight given for non-Ias wrongly classified.

Returns

- **metric_names** (*list*) – Name of elements in metrics: [accuracy, eff, purity, fom]
- **metric_values** (*list*) – list of calculated metrics values for each element

Active Learning loop

Full light curve

<code>learn_loop(nloops, strategy, ...[, ...])</code>	Perform the active learning loop.
---	-----------------------------------

actsnclass.learn_loop

`actsnclass.learn_loop` (*nloops: int, strategy: str, path_to_features: str, output_metrics_file: str, output_queried_file: str, features_method='Bazin', classifier='RandomForest', training='original', batch=1, screen=True*)

Perform the active learning loop. All results are saved to file.

Parameters

- **nloops** (*int*) – Number of active learning loops to run.
- **strategy** (*str*) – Query strategy. Options are 'UncSampling' and 'RandomSampling'.
- **path_to_features** (*str*) – Complete path to input features file.
- **output_metrics_file** (*str*) – Full path to output file to store metric values of each loop.
- **output_queried_file** (*str*) – Full path to output file to store the queried sample.
- **features_method** (*str (optional)*) – Feature extraction method. Currently only 'Bazin' is implemented.

- **classifier** (*str (optional)*) – Machine Learning algorithm. Currently only ‘RandomForest’ is implemented.
- **training** (*str or int (optional)*) – Choice of initial training sample. If ‘original’: begin from the train sample flagged in the file If int: choose the required number of samples at random, ensuring that at least half are SN Ia Default is ‘original’.
- **batch** (*int (optional)*) – Size of batch to be queried in each loop. Default is 1.
- **screen** (*bool (optional)*) – If True, print on screen number of light curves processed.

Time domain

<code>get_original_training(path_to_features[, ...])</code>	Read original full light curve training sample
<code>time_domain_loop(days, output_metrics_file, ...)</code>	Perform the active learning loop.

actsnclass.get_original_training

`actsnclass.get_original_training (path_to_features, method='Bazin', screen=False)`
 Read original full light curve training sample

Parameters

- **path_to_features** (*str*) – Complete path to file holding full light curve features.
- **method** (*str (optional)*) – Feature extraction method. Only option implemented is “Bazin”.
- **screen** (*bool (optional)*) – If true, show on screen comments on the dimensions of some key elements.

Returns Information about the original full light curve analys.

Return type `snactclass.DataBase`

actsnclass.time_domain_loop

`actsnclass.time_domain_loop (days: list, output_metrics_file: str, output_queried_file: str, path_to_features_dir: str, strategy: str, batch=1, canonical=False, classifier='RandomForest', features_method='Bazin', path_to_canonical="", path_to_full_lc_features="", queryable=True, screen=True, training='original')`

Perform the active learning loop. All results are saved to file.

Parameters

- **days** (*list*) – List of 2 elements. First and last day of observations since the beginning of the survey.
- **output_metrics_file** (*str*) – Full path to output file to store metrics for each loop.
- **output_queried_file** (*str*) – Full path to output file to store the queried sample.
- **path_to_features_dir** (*str*) – Complete path to directory holding features files for all days.
- **strategy** (*str*) – Query strategy. Options are ‘UncSampling’ and ‘RandomSampling’.
- **batch** (*int (optional)*) – Size of batch to be queried in each loop. Default is 1.
- **canonical** (*bool (optional)*) – If True, restrict the search to the canonical sample.

- **classifier** (*str (optional)*) – Machine Learning algorithm. Currently only ‘RandomForest’ is implemented.
- **features_method** (*str (optional)*) – Feature extraction method. Currently only ‘Bazin’ is implemented.
- **path_to_canonical** (*str (optional)*) – Path to canonical sample features files. It is only used if “strategy==canonical”.
- **path_to_full_lc_features** (*str (optional)*) – Path to full light curve features file. Only used if training is a number.
- **queryable** (*bool (optional)*) – If True, allow queries only on objects flagged as queryable. Default is True.
- **screen** (*bool (optional)*) – If True, print on screen number of light curves processed.
- **training** (*str or int (optional)*) – Choice of initial training sample. If ‘original’: begin from the train sample flagged in the file If int: choose the required number of samples at random, ensuring that at least half are SN Ia Default is ‘original’.

Plotting

<code>Canvas()</code>	Canvas object, handles and plot information from multiple strategies.
<code>Canvas.load_metrics(path_to_files, ..., [...])</code>	Load and identify set of metrics.
<code>Canvas.set_plot_dimensions()</code>	Set directives for plot sizes.
<code>Canvas.plot_metrics(output_plot_file, ...)</code>	Generate plot for all metrics in files and strategies given as input.

actsnclass.Canvas

class actsnclass.Canvas

Canvas object, handles and plot information from multiple strategies.

Variables

- **axis_label_size** (*int*) – Size of font in axis labels.
- **canonical** (*pd.DataFrame*) – Data from Canonical strategy.
- **fig_size** (*tuple*) – Figure dimensions.
- **rand_sampling** (*pd.DataFrame*) – Data from Random Sampling strategy.
- **tick_label_size** (*int*) – Size of tick labels in both axis.
- **ncolumns** (*int*) – Number of columns in panel grid.
- **nlines** (*int*) – Number of lines in panel grid.
- **nmetrics** (*int*) – Number of metric elements (panels in plot).
- **metrics_names** (*list*) – List of names for metrics to be plotted.
- **unc_sampling** (*pd.DataFrame*) – Data from Uncertainty Sampling strategy.
- **colors** (*dict*) – Colors corresponding to each strategy. They were chosen to follow Mondrian’s color palette. Do not change and try to keep the same palette in adding new elements.

- **labels** (*dict*) – Labels to appear on plot for each strategy.
- **markers** (*dict*) – Plot markers for each strategy.
- **strategies** (*dict*) – Dictionary connecting each data frame to its standard nomenclature (not the plot labels).

load_metrics (*path_to_files: list, strategy_list: list*)

Load metrics and identify set of metrics.

set_plot_dimensions ()

Set directives for plot sizes based on number of metrics.

plot_metrics (*output_plot_file: str, strategies_list: list*)

Generate plot for all metrics in files and strategies given as input.

Examples

Define input variables

```
>>> path_to_files = ['results/metrics_canonical.dat',
>>>                  'results/metrics_random.dat',
>>>                  'results/metrics_unc.dat']
>>> strategies_list = ['Canonical', 'RandomSampling', 'UncSampling']
>>> output_plot = 'plots/metrics1_unc.png'
```

Initiate the Canvas object, read and plot the results for each metric and strategy.

```
>>> cv = Canvas()
>>> cv.load_metrics(path_to_files=path_to_files,
>>>                 strategies_list=strategies_list)
>>> cv.set_plot_dimensions()
>>> cv.plot_metrics(output_plot_file=output_plot,
>>>                 strategies_list=strategies_list)
```

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
<code>load_metrics(path_to_files, strategies_list)</code>	Load and identify set of metrics.
<code>plot_metrics(output_plot_file, strategies_list)</code>	Generate plot for all metrics in files and strategies given as input.
<code>set_plot_dimensions()</code>	Set directives for plot sizes.

actsnclass.Canvas.load_metrics

Canvas.load_metrics (*path_to_files: list, strategies_list: list, metrics_name='snpc'*)

Load and identify set of metrics.

Populates attributes: canonical, unc_sampling or rand_sampling, depending on choice of 'sample'.

Parameters

- **path_to_files** (*str*) – List of paths to metrics files for different strategies.

- **strategies_list** (*list*) – List of all strategies to be included in the same plot. Current possibilities are: ['canonical', 'rand_sampling', 'unc_sampling'].
- **metrics_name** (*str (optional)*) – Identify the metrics to be read. Only option is 'snpc'. Only option is 'snpc'.

actsnclass.Canvas.set_plot_dimensions

Canvas.**set_plot_dimensions** ()

Set directives for plot sizes.

Populates attributes: nmetrics, ncolumns, and fig_size.

actsnclass.Canvas.plot_metrics

Canvas.**plot_metrics** (*output_plot_file: str, strategies_list: list*)

Generate plot for all metrics in files and strategies given as input.

Parameters

- **output_plot_file** (*str*) – Complete path to file to store plot.
- **strategies_list** (*list*) – List of all strategies to be included in the same plot. Current possibilities are: ['canonical', 'rand_sampling', 'unc_sampling'].

Scripts

<code>build_canonical(user_choices)</code>	Build canonical sample for SNPCC data set fitted with Bazin features.
<code>build_time_domain(user_choice)</code>	Generates features files for a list of days of the survey.
<code>fit_dataset(user_choices)</code>	Fit the entire sample with the Bazin function.
<code>make_metrics_plots(user_input)</code>	Generate metric plots.
<code>run_loop(args)</code>	Command line interface to run the active learning loop.
<code>run_time_domain(user_choice)</code>	Command line interface to the Time Domain Active Learning scenario.

actsnclass.build_canonical

actsnclass.**build_canonical** (*user_choices*)

Build canonical sample for SNPCC data set fitted with Bazin features.

Parameters

- **-c** (*bool*) – If True, compute metadata on SNR and peak mag.
- **-d** (*str (optional)*) – Path to raw data directory. Needed only if “compute == False”.
- **-f** (*str*) – Path to features file.
- **-i** (*str (optional)*) – Path to read metadata on SNR and peak mag from file. It is only used if “compute == False”.
- **-m** (*str (optional)*) – Path to output file where to store metadata on SNR and peak mag. Only used if “save == True”.
- **-o** (*str*) – Path to store full canonical sample.

- **-p** (*str (optional)*) – File to store comparison plot. If not provided plot is shown on screen.
- **-s** (*bool*) – If True, save to file metadata on SNR and peakmag.

Examples

Use directly from the command line:

```
>>> build_canonical.py -c <if True compute metadata>
>>>     -d <path to raw data dir>
>>>     -f <input features file> -m <output file for metadata>
>>>     -o <output file for canonical sample> -p <comparison plot file>
>>>     -s <if True save metadata to file>
```

actsnclass.build_time_domain

`actsnclass.build_time_domain(user_choice)`

Generates features files for a list of days of the survey.

Parameters

- **-d** (*sequence*) – Sequence of days since the begin of the survey to be processed.
- **-p** (*str*) – Complete path to raw data directory.
- **-o** (*str*) – Complete path to output time domain directory.

Examples

Use it directly from the command line.

```
>>> build_time_domain.py -d 20 21 22 23 -p <path to raw data dir> -o <path to_
↳output time domain dir>
```

actsnclass.fit_dataset

`actsnclass.fit_dataset(user_choices)`

Fit the entire sample with the Bazin function.

All results are saved to file.

Parameters

- **-dd** (*str*) – Path to directory containing raw data.
- **-o** (*str*) – Path to output feature file.

Examples

Run directly from the command line.

```
>>> fit_dataset.py -dd <path_to_data_dir> -o <output_file>
```

actsnclass.make_metrics_plots

actsnclass.make_metrics_plots(*user_input*)

Generate metric plots.

Parameters

- **-m** (*list*) – List of paths to the metrics files. One path for each learning strategy we wish to plot. This must follow the same order as give in -s.
- **-o** (*str*) – Path to output file where the plot will be stored.
- **-s** (*list*) – List of keywords describing the learning strategies to be plotted. Order must be the same as provided in -m. Options are ['canonical', 'rand_sampling', 'unc_sampling']

Examples

Use it directly from the command line. For example, if you wish to make a metric plot for the random sampling and uncertainty sampling together, do:

```
>>> make_metrics_plots.py -m <path to rand sampling metrics> <path to unc_
    ↳sampling metrics>
>>> -o <path to output plot file> -s RandomSampling UncSampling
```

actsnclass.run_loop

actsnclass.run_loop(*args*)

Command line interface to run the active learning loop.

Parameters

- **-d** (*str*) – Full path to output file to store diagnostics of each loop.
- **-i** (*str*) – Complete path to input features file.
- **-n** (*int*) – Number of active learning loops to run.
- **-q** (*str*) – Full path to output file to store the queried sample.
- **-s** (*str*) – Query strategy. Options are 'UncSampling' and 'RandomSampling'.
- **-b** (*int (optional)*) – Size of batch to be queried in each loop. Default is 1.
- **-c** (*str (optional)*) – Machine Learning algorithm. Currently only 'RandomForest' is implemented.
- **-m** (*str (optional)*) – Feature extraction method. Currently only 'Bazin' is implemented.
- **-t** (*str or int (optional)*) – Choice of initial training sample. If 'original': begin from the train sample flagged in the file If int: choose the required number of samples at random, ensuring that at least half are SN Ia. Default is 'original'.

Examples

Run directly from the command line:

```
>>> run_loop.py -i <input features file> -b <batch size> -n <number of loops>
>>>                 -d <output diagnostic file> -q <output queried sample file>
>>>                 -s <learning strategy> -t <choice of initial training>
```

actsnclass.run_time_domain

actsnclass.run_time_domain(*user_choice*)

Command line interface to the Time Domain Active Learning scenario.

Parameters

- **-d** (*sequence*) – List of 2 elements. First and last day of observations since the beginning of the survey.
- **-m** (*str*) – Full path to output file to store metrics of each loop.
- **-q** (*str*) – Full path to output file to store the queried sample.
- **-f** (*str*) – Complete path to directory holding features files for all days.
- **-s** (*str*) – Query strategy. Options are ‘UncSampling’ and ‘RandomSampling’.
- **-b** (*int (optional)*) – Size of batch to be queried in each loop. Default is 1.
- **-c** (*str (optional)*) – Machine Learning algorithm. Currently only ‘RandomForest’ is implemented.
- **-fm** (*str (optional)*) – Feature extraction method. Currently only ‘Bazin’ is implemented.
- **-sc** (*bool (optional)*) – If True, display comment with size of samples on screen.
- **-t** (*str or int*) – Choice of initial training sample. If ‘original’: begin from the train sample flagged in the file If int: choose the required number of samples at random, ensuring that at least half are SN Ia Default is ‘original’.

Returns

- **metric file** (*file*) – File with metrics calculated in each iteration.
- **queried file** (*file*) – All objects queried during the search, in sequence.

Examples

Use directly from the command line.

```
>>> run_time_domain.py -d <first day of survey> <last day of survey>
>>>                 -m <output metrics file> -q <output queried file> -f <features_
↳directory>
>>>                 -s <learning strategy> -fm <path to full light curve features >
```

Be aware to check the default options as well!

1.4 Indices and tables

- [genindex](#)
- [modindex](#)

- search

Symbols

`__init__()` (*actsnclass.Canonical method*), 23
`__init__()` (*actsnclass.Canvas method*), 38
`__init__()` (*actsnclass.DataBase method*), 29
`__init__()` (*actsnclass.LightCurve method*), 19
`__init__()` (*actsnclass.SNPCCPhotometry method*), 25

A

`accuracy()` (*in module actsnclass*), 34

B

`bazin()` (*in module actsnclass.bazin*), 21
`build_canonical()` (*in module actsnclass*), 39
`build_one_epoch()` (*actsnclass.SNPCCPhotometry method*), 25, 26
`build_samples()` (*actsnclass.DataBase method*), 28, 30
`build_snpcc_canonical()` (*in module actsnclass*), 24
`build_time_domain()` (*in module actsnclass*), 40

C

`Canonical` (*class in actsnclass*), 22
`Canvas` (*class in actsnclass*), 37
`check_queryable()` (*actsnclass.LightCurve method*), 19
`classify()` (*actsnclass.DataBase method*), 28, 30
`create_daily_file()` (*actsnclass.SNPCCPhotometry method*), 25, 26

D

`DataBase` (*class in actsnclass*), 27

E

`efficiency()` (*in module actsnclass*), 34
`errfunc()` (*in module actsnclass.bazin*), 21

`evaluate_classification()` (*actsnclass.DataBase method*), 28, 30

F

`find_neighbors()` (*actsnclass.Canonical method*), 23, 24
`fit_bazin()` (*actsnclass.LightCurve method*), 19, 20
`fit_bazin_all()` (*actsnclass.LightCurve method*), 19, 20
`fit_dataset()` (*in module actsnclass*), 40
`fit_scipy()` (*in module actsnclass.bazin*), 22
`fit_snpcc_bazin()` (*in module actsnclass*), 21
`fom()` (*in module actsnclass*), 34

G

`get_lim_mjds()` (*actsnclass.SNPCCPhotometry method*), 25, 26
`get_original_training()` (*in module actsnclass*), 36
`get_snpcc_metric()` (*in module actsnclass*), 35

L

`learn_loop()` (*in module actsnclass*), 35
`LightCurve` (*class in actsnclass*), 18
`load_bazin_features()` (*actsnclass.DataBase method*), 27, 29
`load_features()` (*actsnclass.DataBase method*), 27, 29
`load_metrics()` (*actsnclass.Canvas method*), 38
`load_snpcc_lc()` (*actsnclass.LightCurve method*), 19, 20

M

`make_metrics_plots()` (*in module actsnclass*), 41
`make_query()` (*actsnclass.DataBase method*), 28, 31

P

`plot_bazin_fit()` (*actsnclass.LightCurve method*), 19, 20

`plot_metrics()` (*actsncclass.Canvas method*), 38, 39
`plot_snpcc_train_canonical()` (*in module actsncclass*), 25
`purity()` (*in module actsncclass*), 34

R

`random_forest()` (*in module actsncclass*), 32
`random_sampling()` (*in module actsncclass*), 33
`run_loop()` (*in module actsncclass*), 41
`run_time_domain()` (*in module actsncclass*), 42

S

`save_metrics()` (*actsncclass.DataBase method*), 28, 31
`save_queried_sample()` (*actsncclass.DataBase method*), 28, 32
`set_plot_dimensions()` (*actsncclass.Canvas method*), 38, 39
`snpcc_get_canonical_info()` (*actsncclass.Canonical method*), 23
`snpcc_identify_samples()` (*actsncclass.Canonical method*), 23, 24
`SNPCCPhotometry` (*class in actsncclass*), 25

T

`time_domain_loop()` (*in module actsncclass*), 36

U

`uncertainty_sampling()` (*in module actsncclass*), 33
`update_samples()` (*actsncclass.DataBase method*), 28, 31