pyActLearn Documentation

Release 0.2.0.dev0

Tinghui Wang

May 20, 2017

Contents

1 Components **3**
1.1 pyActLearn.CASAS 3
1.2 pyActLearn.learning 3
1.3 pyActLearn.performance 3

2 Roadmap **5**
2.1 Installation 6
2.2 pyActLearn.CASAS 7
2.3 pyActLearn.learning 22
2.4 pyActLearn.performance 22
2.5 Reference 25

3 Indices and tables **27**

Bibliography **29**

Python Module Index **31**

pyActLearn is an activity recognition platform designed to recognize ADL (Activities of Daily Living) in smart homes equipped with less intrusive passive monitoring sensors, such as motion detectors, door sensors, thermometers, light switches, etc.

CHAPTER 1

Components

pyActLearn.CASAS

pyActLearn.CASAS contains classes and functions that load and pre-process smart home sensor event data. The pre-processed data are stored in an hdf5 data format with smart home information stored as attributes of the dataset. The processed data are splitted into weeks and days. Class `pyActLearn.CASAS.hdf5.CASASHDF5` can load the hdf5 dataset and use as a feeder for activity recognition learning algorithm.

pyActLearn.learning

pyActLearn.learning contains classes and functions that implement supervised and unsupervised learning algorithms for activity recognition. Some of the classes refers to models provided by other python packages such as hmmlearn (for multinomial hidden markov models) and sklearn (for support vector machine, decision tree, and random forest).

pyActLearn.performance

pyActLearn.performance contains classes and functions that implement multiple performance metrics for activity recognition, including confusion matrix, multi-class classification metrics, event-based scoring, and activity timeliness.

CHAPTER 2

Roadmap

- Data Loading
  - [X] Load event list from legacy raw text files
  - [X] Load event list from csv event files
  - [X] Load sensor information from JSON meta-data file
  - [X] Divide event list by days or weeks
- Pre-processing
  - [X] Statistical feature extraction using sliding window approach
  - [X] Raw interpretation
- Algorithm implementations
  - Supervised Learning
    - [X] Decision Tree
    - [X] HMM
    - [X] SVM
    - [X] Multi-layer Perceptron
    - [X] Stacked De-noising Auto-encoder with fine tuning
    - [X] Recurrent Neural Network with LSTM Cell
    - [ ] Recurrent Neural Network with GRU
  - Un-supervised Learning
    - [X] Stacked De-noising Auto-encoder
    - [X] k-skip-2-gram with Negative Sampling (word2vec)
  - Transfer Learning
- Evaluation

  - [ ] n-Fold Cross-validation
  - [X] Traditional Multi-class Classification Metrics
  - [X] Event-based Continuous Evaluation Metrics
  - [X] Event-based Activity Diagram

- Annotation
  - [X] Back annotate dataset with predicted results
  - [X] Back annotate with probability
- Visualization
  - [X] Sensor distance on floor plan

Installation

Install using pip

You can install pyActLearn package using Python package manager `pip`. The package source codes are avaialble on github.

```
$ pip3 install git+git://github.com/TinghuiWang/pyActLearn.git \
  -r https://raw.githubusercontent.com/TinghuiWang/pyActLearn/master/requirements.txt
```

Note: At the moment, pyActLearn package only supports Python 3. In most Linux distributions, the Python 3 package manager is named as `pip3`.

You can also add `--user` switch to the command to install the package in your home folder, and `--upgrade` switch to pull the latest version from github.

Warning: `pip` may try to install or update packages such as Numpy, Scipy and Theano if they are not present or outdated. If you want to use your system package manager such as `apt` or `yum`, you can add `--no-deps` switch and install all the requirements manually.

Install in Develop Mode

To install the package in developer mode, first fork your own copy of pyActLearn on github. Then, you can clone and edit your repository and install with `setup.py` script using the following command (replace `user` with your own github user name).

```
$ git clone https://github.com/USER/pyActLearn.git
$ python3 setup.py develop
```

pyActLearn.CASAS

CASAS.data

CASAS.data implements `pyActLearn.CASAS.data.CASASData`.

Sensor Event File Format

`pyActLearn.CASAS.data.CASASData` can load the smart home raw sensor event logs in raw text (legacy) format, and comma separated (.csv) files.

Legacy format

Here is a snip of sensor event logs of a smart home in raw text format:

```
2009-06-01 17:51:20.055202   M046    ON
2009-06-01 17:51:22.036689   M046    OFF
2009-06-01 17:51:28.053264   M046    ON
2009-06-01 17:51:30.072223   M046    OFF
2009-06-01 17:51:35.046958   M045    OFF
2009-06-01 17:51:41.096098   M045    ON
2009-06-01 17:51:44.096236   M046    ON
2009-06-01 17:51:45.053722   M045    OFF
2009-06-01 17:51:46.015612   M045    ON
2009-06-01 17:51:47.005712   M046    OFF
2009-06-01 17:51:48.004619   M046    ON
2009-06-01 17:51:49.076356   M046    OFF
2009-06-01 17:51:50.035392   M046    ON
```

The following is an example to load the sensor event logs in legacy text format into class `pyActLearn.CASAS.data.CASASData`.

```
from pyActLearn.CASAS.data import CASASData
data = CASASData(path='twor.summer.2009/annotate')
```

CSV format

Some of the smart home data set are updated to CSV format. Those datasets usually come with meta-data about the smart home including floorplan, sensor location, activities annotated, and other information.

The binary sensor events are logged inside file `event.csv`. Here is a snip of it:

```
2/1/2009,8:00:38 AM,M048,OFF,,,
2/1/2009,8:00:38 AM,M049,OFF,,,
2/1/2009,8:00:39 AM,M028,ON,,,
2/1/2009,8:00:39 AM,M042,ON,,,
2/1/2009,8:00:40 AM,M029,ON,,,
2/1/2009,8:00:40 AM,M042,OFF,,,
2/1/2009,8:00:40 AM,L003,OFF,,,
2/1/2009,8:00:42 AM,M043,OFF,,,
2/1/2009,8:00:42 AM,M037,ON,,,
2/1/2009,8:00:42 AM,M050,OFF,,,
2/1/2009,8:00:42 AM,M044,OFF,,,
2/1/2009,8:00:42 AM,M028,OFF,,,
2/1/2009,8:00:43 AM,M029,OFF,,,
```

The metadata about the smart home is in a json file format. Here is a snip of the metadata for twor dataset:

```
{
"name": "TWOR_2009_test",
"floorplan": "TWOR_2009.png",
"sensors": [
   {
      "name": "M004",
      "type": "Motion",
      "locX": 0.5605087077755726,
      "locY": 0.061440840882448416,
      "sizeX": 0.0222007722007722,
      "sizeY": 0.018656716417910446,
      "description": ""
   },
],
"activities": [
   {
      "name": "Meal Preparation",
      "color": "#FF8A2BE2",
      "is_noise": false,
      "is_ignored": false
   },
]}
```

To load such a dataset, provide the directory path to the constructor of *pyActLearn.CASAS.data.CASASData*.

```
from pyActLearn.CASAS.data import CASASData
data = CASASData(path='twor.summer.2009/')
```

Note: The constructor of *pyActLearn.CASAS.data.CASASData* differentiates the format of sensor log by determining whether the path is a directory or file. If it is a file, it assumes that it is in legacy raw text format. If it is a directory, the constructor looks for `event.csv` file within the directory for binary sensor events, and `dataset.json` for mete-data about the smart home.

Event Pre-processing

Raw sensor event data may need to be pre-processed before the learning algorithm can consume them. For algorithms like Hidden Markov Model, only raw sensor series are needed. For algorithms like decision tree, random forest, multi-layer perceptron, etc., statistic features within a sliding window of fixed length or variable length are calculated. For data used in stacked auto-encoder, the input needs to be normalized between 0 to 1.

pyActLearn.CASAS.data.CASASData.populate_feature() function handles the pre-processing of all binary sensor events. The statistical features implemented in this function includes

- *Window Duration*
- *Last Sensor*
- *Hour of the Event*
- *Seconds of the Event*

- *Sensor Count*
- *Sensor Elapse Time*
- *Dominant Sensor*

Methods to enable and disable specific features or activities are provided as well. Please refer to `pyActLearn.CASAS.data.CASASData` API reference for more information.

Export Data

After the data are pre-processed, the features and labels can be exported to excel file (.xlsx) via function `pyActLearn.CASAS.data.CASASData.write_to_xlsx()`.

`pyActLearn.CASAS.data.CASASData.export_hdf5()` will save the pre-processed features and target labels in hdf5 format. The meta-data is saved as attributes of the root node of hdf5 dataset. The hdf5 file can be viewed using hdfviewer.

Here is an example loading raw sensor events and save to hdf5 dataset file.

```
from pyActLearn.CASAS.data import CASASData
data = CASASData(path='datasets/twor.2009/')
data.populate_feature(method='stat', normalized=True, per_sensor=True)
data..export_hdf5(filename='hdf5/twor_2009_stat.hdf5', comments='')
```

API Reference

class `pyActLearn.CASAS.data.`**`CASASData`**`(`*path*`)`

Bases: `object`

A class to load activity data from CASAS smart home datasets.

The class load raw activity sensor events from CASAS smart home datasets. The class provides methods to pre-process the data for future learning algorithms for activity recognition. The pre-processed data can be exported to xlsx files for verification, and hdf5 file for faster read and search when evaluating a activity recognition algorithm.

Parameters **`path`** (`str`) – path to a dataset directory, the dataset event.rst file for dataset in legacy format.

Variables

- **`sensor_list`** (`dict`) – A dictionary containing sensor information.
- **`activity_list`** (`dict`) – A dictionary containing activity information.
- **`event_list`** (`list` of `dict`) – List of data used to store raw events.
- **`x`** (`numpy.ndarray`) – 2D numpy array that contains calculated feature data.
- **`y`** (`numpy.ndarray`) – 2D numpy array that contains activity label corresponding to feature array
- **`data_path`** (`str`) – path to data file.
- **`home`** (`pyActLearn.CASAS.home.CASASHome`) – `CASAS.home.CASASHome` object that stores the home information associated with the dataset.
- **`is_legacy`** (`bool`) – Defaults to False. If the dataset loaded is in legacy format or not.
- **`is_stat_feature`** (`bool`) – Calculate statistical features or use raw data in `x`

- **is_labeled** (`bool`) – If given dataset is labeled
- **time_list** (`list` of `datetime.datetime`) – Datetime of each entry in `x`. Used for back annotation, and splitting dataset by weeks or days.
- **feature_list** (`dict`) – A dictionary of statistical features used in statistical feature calculation
- **routines** (`dict`) – Function routines that needs to run every time when calculating features. Excluded from pickling.
- **num_enabled_features** (`int`) – Number of enabled features.
- **num_static_features** (`int`) – Number of features related to window
- **num_per_sensor_features** (`int`) – Number of features that needs to be calculated per enabled sensor
- **events_in_window** (`int`) – Number of sensor events (or statistical features of a sliding window) grouped in a feature vector.

disable_activity (*activity_label*)
Disable an activity

Parameters **activity_label** (`str`) – Activity label

disable_feature (*feature_name*)
Disable a feature

Parameters **feature_name** (`str`) – Feature name.

disable_routine (*routine*)
Disable a routine

Check all enabled feature list and see if the routine is used by other features. If no feature need the routine, disable it

Parameters **routine** (*pyActLearn.CASAS.stat_features.FeatureRoutineTemplate*) – routine to be disabled

disable_sensor (*sensor_name*)
Disable a sensor

Parameters **sensor_name** (`str`) – Sensor Name

enable_activity (*activity_label*)
Enable an activity

Parameters **activity_label** (`str`) – Activity label

Returns The index of the enabled activity

Return type `int`

enable_feature (*feature_name*)
Enable a feature

Parameters **feature_name** (`str`) – Feature name.

enable_routine (*routine*)
Enable a given routine

Parameters **routine** (*pyActLearn.CASAS.stat_features.FeatureRoutineTemplate*) – routine to be disabled

enable_sensor (*sensor_name*)
Enable a sensor

Parameters **sensor_name** (str) – Sensor Name

Returns int: The index of the enabled sensor

export_fuel (*directory*, *break_by='week'*, *comments=''*)
Export feature and label vector into hdf5 file and store the class information in a pickle file

Parameters

- **directory** (str) – The directory to save hdf5 and complementary dataset information
- **break_by** (str) – Select the way to split the data, either by `'week'` or `'day'`
- **comments** (str) – Additional comments to add

export_hdf5 (*filename*, *comments=''*, *bg_activity='Other_Activity'*, *driver=None*)
Export the dataset into a hdf5 dataset file with meta-data logged in attributes.

To load the data, you can use `pyActLearn.CASAS.h5py.CASASH5PY` class.

Parameters

- **filename** (str) – The directory to save hdf5 and complementary dataset information.
- **comments** (str) – Additional comments to add.
- **bg_activity** (str) – Background activity label.
- **driver** (str) – h5py dataset R/W driver.

get_activities_by_indices (*activity_ids*)
Get a group of activities by their corresponding indices

Parameters **activity_ids** (list of int) – A list of activity indices

Returns A list of activity labels in the same order

Return type list of str

get_activity_by_index (*activity_id*)
Get Activity name by their index

Parameters **activity_id** (int) – Activity index

Returns Activity label

Return type str

get_activity_color (*activity_label*)
Find the color string for the activity.

Parameters **activity_label** (str) – activity label

Returns RGB color string

Return type str

get_activity_index (*activity_label*)
Get Index of an activity

Parameters **activity_label** (str) – Activity label

Returns Activity index (-1 if not found or not enabled)

Return type int

get_enabled_activities()
Get label list of all enabled activities

Returns list of activity labels

Return type `list` of `str`

get_enabled_sensors()
Get the names of all enabled sensors

Returns List of sensor names

Return type `list` of `str`

get_feature_by_index(*index*)
Get Feature Name by Index

Parameters **index** (`int`) – column index of feature

Returns

(feature name, sensor name) tuple. If it is not per-sensor feature, the sensor name is None.

Return type `tuple` of `str`

get_feature_string_by_index(*index*)
Get the string describing the feature specified by column index

Parameters **index** (`int`) – column index of feature

Returns Feature string

Return type `str`

get_sensor_by_index(*sensor_id*)
Get the name of sensor by index

Parameters **sensor_id** (`int`) – Sensor index

Returns Sensor name

Return type `str`

get_sensor_index(*sensor_name*)
Get Sensor Index

Parameters **sensor_name** (`str`) – Sensor Name

Returns Sensor index (-1 if not found or not enabled)

Return type `int`

populate_feature(*method='raw'*, *normalized=True*, *per_sensor=True*)
Populate the feature vector in `x` and activities in *y*

Parameters

- **method** (`str`) – The method to convert sensor events into feature vector. Available methods are `'raw'` and `'stat'`.
- **normalized** (`bool`) – Will each feature be normalized between 0 and 1?
- **per_sensor** (`bool`) – For features related with sensor ID, are they

summary()
Print summary of loaded datasets

write_to_xlsx (*filename*, *start=0*, *end=-1*)
Write to file in xlsx format

Parameters

- **filename** (`str`) – xlsx file name.
- **start** (`int`) – start index.
- **end** (`int`) – end index.

CASAS.h5py

casas_hdf5_doc_master implements *pyActLearn.CASAS.h5py.CASASHDF5*.

Dataset Structure

HDF5 is a data model, library and file format for storing and managing data. h5py package is the python interface to read and write HDF5 file. You can open and view the HDF5 file using hdfviewer.

The pre-processed feature array `x` is stored as dataset `/features`. Corresponding target labels is stored as dataset `/targets`. The corresponding time for each entry is stored at `/time` as array of bytes (HDF5 does not support `str`).

The meta-data of the smart home is stored as attributes of the root node. The table below summarizes the description of all those attributes.

Attribute	Description
bg_target	Name of background activity.
comment	Description of the dataset.
days	List of start and stop index tuple of each segment when the dataset is splitted by days.
weeks	List of start and stop index tuple of each segment when the dataset is splitted by weeks.
features	Feature name corresponding to each column in `/features` dataset.
targets	List of activity labels.
target_color	List of color string for each activity for visualization.
sources	List of dataset names in the file.
sensors	List of sensor names

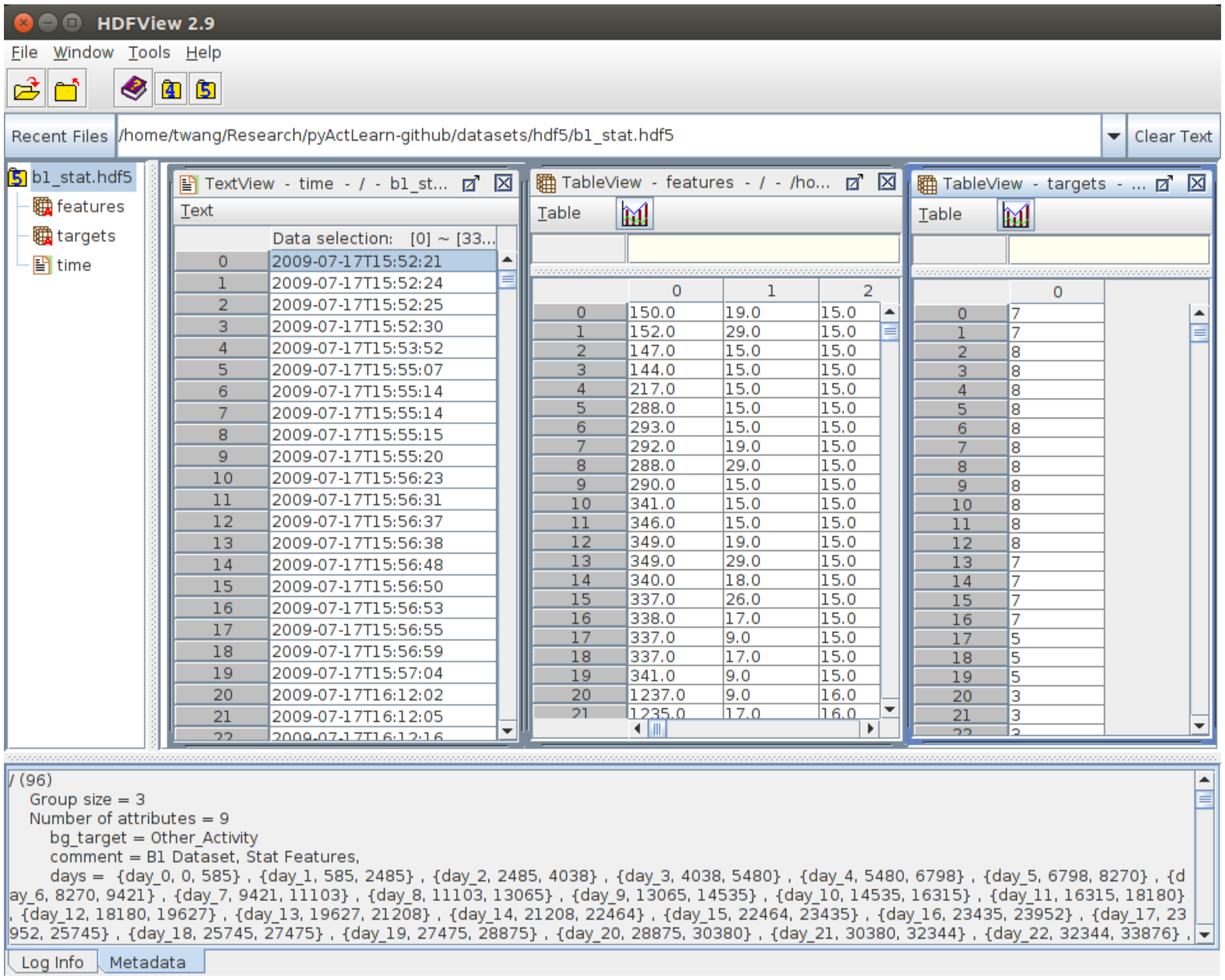
The image below gives a glimpse of the hdf5 structure in hdfviewer.

Load and Fetch Data from HDF5

pyActLearn.CASAS.h5py.CASASHDF5 provides multiple interfaces for accessing and loading the data from hdf5 file. The dataset is usually split by weeks and days. Function *pyActLearn.CASAS.h5py.CASASHDF5.fetch_data()* will load the time, features and target labels of the time frame provided via the start split and end split names.

Here is the code snip to load the data from splits to train a support vector machine.

```
import sklearn.svm
from pyActLearn.CASAS.h5py import CASASHDF5
# Load dataset
ds = CASASHDF5(path='twor_statNormPerSensor.hdf5')
# Training
time, feature, target = ds.fetch_data(start_split='week_1', stop_split='week_4')
x = feature
y = target.flatten().astype(np.int)
```



/home/twang/Research/pyActLearn-github/datasets/hdf5/b1_stat.hdf5

	Data selection: [0] ~ [33...
0	2009-07-17T15:52:21
1	2009-07-17T15:52:24
2	2009-07-17T15:52:25
3	2009-07-17T15:52:30
4	2009-07-17T15:53:52
5	2009-07-17T15:55:07
6	2009-07-17T15:55:14
7	2009-07-17T15:55:14
8	2009-07-17T15:55:15
9	2009-07-17T15:55:20
10	2009-07-17T15:56:23
11	2009-07-17T15:56:31
12	2009-07-17T15:56:37
13	2009-07-17T15:56:38
14	2009-07-17T15:56:48
15	2009-07-17T15:56:50
16	2009-07-17T15:56:53
17	2009-07-17T15:56:55
18	2009-07-17T15:56:59
19	2009-07-17T15:57:04
20	2009-07-17T16:12:02
21	2009-07-17T16:12:05
22	2009-07-17T16:12:16

	0	1	2
0	150.0	19.0	15.0
1	152.0	29.0	15.0
2	147.0	15.0	15.0
3	144.0	15.0	15.0
4	217.0	15.0	15.0
5	288.0	15.0	15.0
6	293.0	15.0	15.0
7	292.0	19.0	15.0
8	288.0	29.0	15.0
9	290.0	15.0	15.0
10	341.0	15.0	15.0
11	346.0	15.0	15.0
12	349.0	19.0	15.0
13	349.0	29.0	15.0
14	340.0	18.0	15.0
15	337.0	26.0	15.0
16	338.0	17.0	15.0
17	337.0	9.0	15.0
18	337.0	17.0	15.0
19	341.0	9.0	15.0
20	1237.0	9.0	16.0
21	1235.0	17.0	16.0

	0
0	7
1	7
2	8
3	8
4	8
5	8
6	8
7	8
8	8
9	8
10	8
11	8
12	8
13	7
14	7
15	7
16	7
17	5
18	5
19	5
20	3
21	3
22	3

```
/ (96)
    Group size = 3
    Number of attributes = 9
        bg_target = Other_Activity
        comment = B1 Dataset, Stat Features,
        days =  {day_0, 0, 585} , {day_1, 585, 2485} , {day_2, 2485, 4038} , {day_3, 4038, 5480} , {day_4, 5480, 6798} , {day_5, 6798, 8270} , {d
ay_6, 8270, 9421} , {day_7, 9421, 11103} , {day_8, 11103, 13065} , {day_9, 13065, 14535} , {day_10, 14535, 16315} , {day_11, 16315, 18180}
, {day_12, 18180, 19627} , {day_13, 19627, 21208} , {day_14, 21208, 22464} , {day_15, 22464, 23435} , {day_16, 23435, 23952} , {day_17, 23
952, 25745} , {day_18, 25745, 27475} , {day_19, 27475, 28875} , {day_20, 28875, 30380} , {day_21, 30380, 32344} , {day_22, 32344, 33876} ,
```

Fig. 2.1: Smart home pre-processed data in hdf5 format.

```
model = sklearn.svm.SVC(kernel='rbf')
model.fit(x, y)
# Testing
time, feature, target = ds.fetch_data(start_split='week_1', stop_split='week_4')
x = feature
y = model.predict(x)
```

API Reference

class `pyActLearn.CASAS.h5py.`**`CASASHDF5`**(*filename*, *mode='r'*, *driver=None*)

Bases: `object`

CASASHDF5 Class to create and retrieve CASAS smart home data from h5df file

The data saved to or retrieved from a H5PY data file are pre-calculated features by `CASASData` class. The H5PY data file also contains meta-data about the dataset, which include description for each feature, splits by week and/or splits by days.

Variables **`_file`** (`h5py.File`) – `h5py.File` object that represents root group.

Parameters

- **`filename`** (`str`) – HDF5 File Name
- **`mode`** (`str`) – 'r' for load from the file, and 'w' for create a new h5py data

`close`()

Close Dataset

`create_comments`(*comment*)

Add comments to dataset

Parameters **`comment`** (`str`) – Comments to the dataset

`create_features`(*feature_array*, *feature_description*)

Create Feature Dataset

Parameters

- **`feature_array`** (`numpy.ndarray`) – Numpy array holding calculated feature vectors
- **`feature_description`** (`list` of `str`) – List of strings that describe each column of feature vectors.

`create_sensors`(*sensors*)

Add sensors list to attributes

If the sensor IDs in the dataset is not binary coded, there is a need to provide the sensor list to go along with the feature vectors.

Parameters **`sensors`** (`list` of `str`) – List of sensor name corresponds to the id in the feature array.

`create_splits`(*days*, *weeks*)

Create splits by days and weeks

Parameters

- **`days`** (`list` of `int`) – Start index for each day
- **`weeks`** (`list` of `int`) – Start index for week

create_targets(*target_array*, *target_description*, *target_colors*)
Create Target Dataset

Parameters

- **target_array** (numpy.ndarray) – Numpy array holding target labels
- **target_description** (list of str) – List of strings that describe each each target class.
- **target_colors** (list of str) – List of color values corresponding to each target class.

create_time_list(*time_array*)
Create Time List

Parameters **time_array** (list of datetime) – datetime corresponding to each feature vector in feature dataset.

fetch_data(*start_split=None*, *stop_split=None*, *pre_load=0*)
Fetch data between start and stop splits

Parameters

- **start_split** (str) – Begin of data
- **stop_split** (str) – End of data
- **pre_load** (int) – Load extra number of data before start split.

Returns

Returns a tuple of all sources sliced by the split defined. The sources should be in the order of ('time', 'feature', 'target')

Return type tuple of numpy.ndarray

flush()
Write To File

get_bg_target()
Get the description of the target class considered background in the dataset.

Returns Name of the class which is considered background in the dataset. Usually it is 'Other_Activity'.

Return type str

get_bg_target_id()
Get the id of the target class considered background.

Returns The index of the target class which is considered background in the dataset.

Return type int

get_days_info()
Get splits by day.

Returns

List of (key, value) tuple, where key is the name of the split and value is number of items in that split.

Return type List of tuple

get_feature_description_by_index(*i*)
Get the description of feature column i.

Parameters **i** (`int`) – Column index.

Returns Corresponding column description.

Return type `str`

`get_sensor_by_index`(*i*)
Get sensor name by index

Parameters **i** (`int`) – Index to sensor

`get_target_color_by_index`(*i*)
Get the color string of target class i.

Parameters **i** (`int`) – Class index.

Returns Corresponding target class color string.

Return type `str`

`get_target_description_by_index`(*i*)
Get target description by class index i.

Parameters **i** (`int`) – Class index.

Returns Corresponding target class description.

Return type `str`

`get_target_descriptions`()
Get list of target descriptions

Returns List of target class description strings.

Return type `list` of `str`

`get_weeks_info`()
Get splits by week.

Returns

List of (key, value) tuple, where key is the name of the split and value is number of items in that split.

Return type `List` of `tuple`

`is_bg_target`(*i=None*, *label=None*)
Check if the target class given by **:param:‘i‘** or **:param:‘label‘** is considered background

Parameters

- **i** (`int`) – Class index.
- **label** (`str`) – Class name.

Returns True if it is considered background.

Return type `bool`

`num_between_splits`(*start_split=None*, *stop_split=None*)
Get the number of item between splits

Parameters

- **start_split** (`str`) – Begin of data
- **stop_split** (`str`) – End of data

Returns The number of items between two splits.

Return type int

num_features()
Get number of features in the dataset

num_sensors()
Return the number of sensors in the sensor list

num_targets()
Total number of target classes.

Returns Total number of target classes.

Return type int

set_background_target(*target_name*)
Set 'target_name' as background target

Parameters **target_name** (str) – Name of background target

Statistical Features

For activity recognition based on learning algorithms like support vector machine (SVM), decision tree, random forest, multi-layer perceptron. *[Krishnan2014]* investigated various sliding window approaches to generate such statistical features.

Window Duration

class pyActLearn.CASAS.stat_features.**WindowDuration**(*normalized=False*)
Bases: *pyActLearn.CASAS.stat_features.FeatureTemplate*

Length of the window.

Any sliding window should have a duration of less than half a day. If it is, it is probable that there some missing sensor events, so the statistical features calculated for such a window is invalid.

Parameters **normalized** (bool) – If true, the hour is normalized between 0 to 1.

get_feature_value(*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)
Get the duration of the window in seconds. Invalid if the duration is greater than half a day.

Note: Please refer to *get_feature_value()* for information about parameters.

Last Sensor

class pyActLearn.CASAS.stat_features.**LastSensor**(*per_sensor=False*)
Bases: *pyActLearn.CASAS.stat_features.FeatureTemplate*

Sensor ID of the last sensor event.rst of the window.

For algorithms like decision trees and hidden markov model, sensor ID can be directly used as features. However, in other algorithms such as multi-layer perceptron, or support vector machine, the sensor ID needs to be binary coded.

Parameters **per_sensor** (bool) – True if the sensor ID needs to be binary coded.

get_feature_value (*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)

Get the sensor which fired the last event.rst in the sliding window.

If it is configured as per-sensor feature, it returns 1 if the sensor specified triggers the last event.rst in the window. Otherwise returns 0. If it is configured as a non-per-sensor feature, it returns the index of the index corresponding to the dominant sensor name that triggered the last event.rst.

Note: Please refer to `get_feature_value()` for information about parameters.

Hour of the Event

class `pyActLearn.CASAS.stat_features.`**EventHour** (*normalized=False*)

Bases: `pyActLearn.CASAS.stat_features.FeatureTemplate`

Hour of last event.rst.

It returns the hour of last sensor event.rst in the sliding window. If `normalized` is set to `True`, the hour is divided by 24, so that the value is bounded between 0 to 1.

Parameters **normalized** (`bool`) – If true, the hour is normalized between 0 to 1.

get_feature_value (*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)

Get the hour when the last sensor event.rst in the window occurred

Note: Please refer to `get_feature_value()` for information about parameters.

Seconds of the Event

class `pyActLearn.CASAS.stat_features.`**EventSeconds** (*normalized=False*)

Bases: `pyActLearn.CASAS.stat_features.FeatureTemplate`

Seconds of last event.rst.

The time of the hour (in seconds) of the last sensor event.rst in the window. If `normalized` is `True`, the seconds is divided by 3600.

Parameters **normalized** (`bool`) – If true, the hour is normalized between 0 to 1.

get_feature_value (*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)

Get the time within an hour when the last sensor event.rst in the window occurred (in seconds)

Note: Please refer to `get_feature_value()` for information about parameters.

Sensor Count

class `pyActLearn.CASAS.stat_features.`**SensorCount** (*normalized=False*)

Bases: `pyActLearn.CASAS.stat_features.FeatureTemplate`

Counts the occurrence of each sensor within the sliding window.

The count of occurrence of each sensor is normalized to the length (total number of events) of the window if the `normalized` is set to True.

Parameters **`normalized`** (`bool`) – If true, the count of each sensor is normalized between 0 to 1.

`get_feature_value`(*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)
Counts the number of occurrence of the sensor specified in current window.

Sensor Elapse Time

class `pyActLearn.CASAS.stat_features.`**`SensorElapseTime`**(*normalized=False*)
Bases: *`pyActLearn.CASAS.stat_features.FeatureTemplate`*

The time elapsed since last firing (in seconds)

`get_feature_value`(*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)
Get elapse time of specified sensor in seconds

Dominant Sensor

class `pyActLearn.CASAS.stat_features.`**`DominantSensor`**(*per_sensor=False*)
Bases: *`pyActLearn.CASAS.stat_features.FeatureTemplate`*

Dominant Sensor of current window.

The sensor that fires the most amount of sensor event.rst in the current window.

Parameters **`per_sensor`** (`bool`) – True if the sensor ID needs to be binary coded.

`get_feature_value`(*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)
If per_sensor is True, returns 1 with corresponding sensor Id. otherwise, return the index of last sensor in the window

Feature Template

class `pyActLearn.CASAS.stat_features.`**`FeatureTemplate`**(*name*, *description*, *enabled=True*, *normalized=True*, *per_sensor=False*, *routine=None*)
Bases: `object`

Statistical Feature Template

Parameters

- **`name`** (`str`) – Feature name.
- **`description`** (`str`) – Feature description.
- **`per_sensor`** (`bool`) – If the feature is calculated for each sensor.
- **`enabled`** (`bool`) – If the feature is enabled.
- **`routine`** (*`FeatureRoutineTemplate`*) – Routine structure.
- **`normalized`** (`bool`) – If the value of feature needs to be normalized.

Variables

- **`name`** (`str`) – Feature name.
- **`description`** (`str`) – Feature description.
- **`index`** (`int`) – Feature index.
- **`normalized`** (`bool`) – If the value of feature needs to be normalized.

- **per_sensor** (bool) – If the feature is calculated for each sensor.
- **enabled** (bool) – If the feature is enabled.
- **routine** (*FeatureRoutineTemplate*) – Routine structure.
- **_is_value_valid** (bool) – If the value calculated is valid

get_feature_value (*data_list*, *cur_index*, *window_size*, *sensor_info*, *sensor_name=None*)
Abstract method to get feature value

Parameters

- **data_list** (list) – List of sensor data.
- **cur_index** (int) – Index of current data record.
- **window_size** (int) – Sliding window size.
- **sensor_info** (dict) – Dictionary containing sensor index information.
- **sensor_name** (str) – Sensor Name.

Returns feature value

Return type double

is_value_valid
Statistical feature value valid check

Due to errors and failures of sensors, the statistical feature calculated may go out of bound. This abstract method is used to check if the value calculated is valid. If not, it will not be inserted into feature vectors.

Returns True if the result is valid.

Return type bool

Feature Update Routine Template

class pyActLearn.CASAS.stat_features.**FeatureRoutineTemplate** (*name*, *description*, *enabled=True*)

Bases: object

Feature Routine Class

A routine that calculate statistical features every time the window slides.

Variables

- **name** (str) – Feature routine name.
- **description** (str) – Feature routine description.
- **enabled** (str) – Feature routine enable flag.

clear ()
Clear Internal Data Structures if recalculation is needed

update (*data_list*, *cur_index*, *window_size*, *sensor_info*)
Abstract update method

For some features, we will update some statistical data every time we move forward a data record, instead of going back through the whole window and try to find the answer. This function will be called every time we advance in data record.

Parameters

- **data_list** (list) – List of sensor data.
- **cur_index** (int) – Index of current data record.
- **window_size** (int) – Sliding window size.
- **sensor_info** (dict) – Dictionary containing sensor index information.

pyActLearn.learning

pyActLearn.performance

Multi-Class Performance Metrics

In most literature, standard multi-class performance metrics are used to evaluate an activity recognition algorithm. In module `pyActLearn.performace`, the following functions get the confusion matrix and calculate per-class performance and overall micro and macro performance.

`pyActLearn.performance.`**`get_confusion_matrix`**(*num_classes*, *label*, *predicted*)

Calculate confusion matrix based on ground truth and predicted result

Parameters

- **num_classes** (int) – Number of classes
- **label** (list of int) – ground truth labels
- **predicted** (list of int) – predicted labels

Returns Confusion matrix (*numpy_class* by *numpy_class*)

Return type `numpy.array`

`pyActLearn.performance.`**`get_performance_array`**(*confusion_matrix*)

Calculate performance matrix based on the given confusion matrix

[Sokolova2009] provides a detailed analysis for multi-class performance metrics.

Per-class performance metrics:

0.**True_Positive**: number of samples that belong to class and classified correctly

1.**True_Negative**: number of samples that correctly classified as not belonging to class

2.**False_Positive**: number of samples that belong to class and not classified correctMeasure:

3.**False_Negative**: number of samples that do not belong to class but classified as class

4.**Accuracy**: Overall, how often is the classifier correct? (TP + TN) / (TP + TN + FP + FN)

5.**Misclassification**: Overall, how often is it wrong? (FP + FN) / (TP + TN + FP + FN)

6.**Recall**: When it's actually yes, how often does it predict yes? TP / (TP + FN)

7.**False Positive Rate**: When it's actually no, how often does it predict yes? FP / (FP + TN)

8.**Specificity**: When it's actually no, how often does it predict no? TN / (FP + TN)

9.**Precision**: When it predicts yes, how often is it correct? TP / (TP + FP)

10.**Prevalence**: How often does the yes condition actually occur in our sample? Total(class) / Total(samples)

11.**F(1) Measure**: 2 * (precision * recall) / (precision + recall)

12. **G Measure**: sqrt(precision * recall)

Gets Overall Performance for the classifier

0. **Average Accuracy**: The average per-class effectiveness of a classifier
1. **Weighed Accuracy**: The average effectiveness of a classifier weighed by prevalence of each class
2. **Precision (micro)**: Agreement of the class labels with those of a classifiers if calculated from sums of per-text decision
3. **Recall (micro)**: Effectiveness of a classifier to identify class labels if calculated from sums of per-text decisions
4. **F-Score (micro)**: Relationship between data's positive labels and those given by a classifier based on a sums of per-text decisions
5. **Precision (macro)**: An average per-class agreement of the data class labels with those of a classifiers
6. **Recall (macro)**: An average per-class effectiveness of a classifier to identify class labels
7. **F-Score (micro)**: Relations between data's positive labels and those given by a classifier based on a per-class average
8. **Exact Matching Ratio**: The average per-text exact classification

Note: In Multi-class classification, Micro-Precision == Micro-Recall == Micro-FScore == Exact Matching Ratio (Multi-class classification: each input is to be classified into one and only one class)

Parameters

- **`num_classes`** (`int`) – Number of classes
- **`confusion_matrix`** (`numpy.array`) – Confusion Matrix (numpy array of num_class by num_class)

Returns tuple of overall performance and per class performance

Return type `tuple` of `numpy.array`

Event-based Scoring

[Minnen2006] and *[Ward2011]* proposed a set of performance metrics and visualizations for continuous activity recognition. In both papers, the authors examine the issues in continuous activity recognition and argued that the traditional standard multi-class evaluation methods fail to capture common artefacts found in continuous AR.

In both papers, the false positives and false negatives are further divided into six categories to faithfully capture the nature of those errors in the context of continuous AR.

Whenever an error occurs, it is both a false positive with respect to the prediction label and a false negative with respect to the ground truth label.

False positive errors are divided into the following three categories:

- Insertion (I): A FP that corresponds exactly to an inserted return.
- Merge (M): A FP that occurs between two TP segments within a merge return.
- Overfill (O): A FP that occurs at the start or end of a partially matched return.

False negatives errors are divided into the following three categories:

- Deletion (D): A FN that corresponds exactly to a deleted event.

- Fragmenting (F): A FN that corresponds exactly to a deleted event.
- Underfill (U): A FN that occurs at the start or end of a detected event.

API Reference

`pyActLearn.performance.event.`**`score_segment`**(*truth*, *prediction*, *bg_label=-1*)

Score Segments

According to *[Minnen2006]* and *[Ward2011]*, a segment is defined as the largest part of an event on which the comparison between the ground truth and the output of recognition system can be made in an unambiguous way. However, in this piece of code, we remove the limit where the segment is the largest part of an event. As long as there is a match between prediction and ground truth, it is recognized as a segment.

There are four possible outcomes to be scored: TP, TN, FP and FN. In event-based performance scoring, the FP and FN are further divided to the following cases:

- Insertion (I): A FP that corresponds exactly to an inserted return.
- Merge (M): A FP that occurs between two TP segments within a merge return.
- Overfill (O): A FP that occurs at the start or end of a partially matched return.
- Deletion (D): A FN that corresponds exactly to a deleted evjmk, ent.
- Fragmenting (F): A FN that corresponds exactly to a deleted event.
- Underfill (U): A FN that occurs at the start or end of a detected event.

Parameters

- **`truth`** (`numpy.ndarray`) – Ground truth
- **`prediction`** (`numpy.ndarray`) – prediction
- **`bg_label`** (`numpy.ndarray`) – Background label

Returns An array with truth and event-based scoring labels

Return type `numpy.ndarray`

`pyActLearn.performance.event.`**`per_class_event_scoring`**(*num_classes*, *truth*, *prediction*, *truth_scoring*, *prediction_scoring*)

Create per-class event scoring to identify the contribution of event-based errors to the traditional recall and false-positive rate.

Instead of doing an EAD as proposed in previous two papers, we look at **Recall** and **FPR** separately.

Recall is defined as TP/(TP + FN). In another word, how often does it predict yes when it's actually yes? The errors in the false negatives, such as Deletion, Fragmenting, and Underfill, adds up to the FP. A Deletion means a total miss of an activity. Underfill represents an error on the begin and end boundary of the event. Fragmenting represents a glitch in the prediction.

Precision is defined as TP/(TP + FP). In another word, how often is it a yes when it is predicted yes? The error in the false positives, such as Insertion, Merge and Overfill, adds up to the FP. In the task of ADL recognition, insertion may be caused by human error in labeling. Overfill represents a disagreement of the begin/end boundary of an activity, but the merge is a glitch in the prediction.

The function goes through the scoring of prediction and ground truth - and returns two dictionary that summaries the contribution of all those errors to **Recall** and **False Positive Rate** scores.

Parameters

- **num_classes** (`int`) – Total number of target classes
- **truth** (`numpy.ndarray`) – Ground truth array, shape (num_samples,)
- **prediction** (`numpy.ndarray`) – Prediction array, shape (num_samples,)
- **truth_scoring** (`numpy.ndarray`) – Event scoring with respect to ground truth labels (i.e. false negatives are further divided into Deletion, Fragmenting, and Underfill). The information in this array is used to fill **Recall** measurement.
- **prediction_scoring** (`numpy.ndarray`) – Event scoring with respect to prediction labels (i.e. false positives are further divided into Insertion, Merging and Overfill). The information in this array is used to fill **Precision** measurement.

Returns Tuple of event-based scoring summarie for recall and precision. Each summary array has a shape of (num_classes,).

Return type `tuple` of `numpy.ndarray`

`pyActLearn.performance.event.`**`per_class_segment_scoring`**(*num_classes*, *truth*, *prediction*, *truth_scoring*, *prediction_scoring*)

Create per-class event scoring to identify the contribution of event-based errors to the traditional recall and false-positive rate. The count is based on each event segment instead of each sensor event.

Parameters

- **num_classes** (`int`) – Total number of target classes
- **truth** (`numpy.ndarray`) – Ground truth array, shape (num_samples,)
- **prediction** (`numpy.ndarray`) – Prediction array, shape (num_samples,)
- **truth_scoring** (`numpy.ndarray`) – Event scoring with respect to ground truth labels (i.e. false negatives are further divided into Deletion, Fragmenting, and Underfill). The information in this array is used to fill **Recall** measurement.
- **prediction_scoring** (`numpy.ndarray`) – Event scoring with respect to prediction labels (i.e. false positives are further divided into Insertion, Merging and Overfill). The information in this array is used to fill **Precision** measurement.

Returns Tuple of event-based scoring summarie for recall and precision. Each summary array has a shape of (num_classes,).

Return type `tuple` of `numpy.ndarray`

Reference

Activity Recognition

Performance Metrics

CHAPTER 3

Indices and tables

- genindex
- modindex

Bibliography

[Cook2009] Cook, D.J. and Schmitter-Edgecombe, M., 2009. Assessing the quality of activities in a smart environment. Methods of information in medicine, 48(5), p.480.

[Kim2010] Kim, E., Helal, S. and Cook, D., 2010. Human activity recognition and pattern discovery. IEEE Pervasive Computing, 9(1).

[Cook2012] Cook, D.J., 2012. Learning setting-generalized activity models for smart spaces. IEEE intelligent systems, 27(1), pp.32-38.

[Krishnan2014] Krishnan, N.C. and Cook, D.J., 2014. Activity recognition on streaming sensor data. Pervasive and mobile computing, 10, pp.138-154.

[Minnen2006] Minnen, D., Westeyn, T., Starner, T., Ward, J. and Lukowicz, P., 2006. Performance metrics and evaluation issues for continuous activity recognition. Performance Metrics for Intelligent Systems, 4.

[Sokolova2009] Sokolova, M. and Lapalme, G., 2009. A systematic analysis of performance measures for classification tasks. Information Processing & Management, 45(4), pp.427-437.

[Ward2011] Ward, J.A., Lukowicz, P. and Gellersen, H.W., 2011. Performance metrics for activity recognition. ACM Transactions on Intelligent Systems and Technology (TIST), 2(1), p.6.

[Hammerla2015] Hammerla, N.Y. and Plötz, T., 2015, September. Let's (not) stick together: pairwise similarity biases cross-validation in activity recognition. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (pp. 1041-1051). ACM.

Python Module Index

p

pyActLearn.CASAS.data, 9
pyActLearn.CASAS.h5py, 15

Index

C

CASASData (class in pyActLearn.CASAS.data), 9
CASASHDF5 (class in pyActLearn.CASAS.h5py), 15
clear() (pyActLearn.CASAS.stat_features.FeatureRoutineTemplate method), 21
close() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_comments() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_features() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_sensors() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_splits() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_targets() (pyActLearn.CASAS.h5py.CASASHDF5 method), 15
create_time_list() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16

D

disable_activity() (pyActLearn.CASAS.data.CASASData method), 10
disable_feature() (pyActLearn.CASAS.data.CASASData method), 10
disable_routine() (pyActLearn.CASAS.data.CASASData method), 10
disable_sensor() (pyActLearn.CASAS.data.CASASData method), 10
DominantSensor (class in pyActLearn.CASAS.stat_features), 20

E

enable_activity() (pyActLearn.CASAS.data.CASASData method), 10
enable_feature() (pyActLearn.CASAS.data.CASASData method), 10
enable_routine() (pyActLearn.CASAS.data.CASASData method), 10
enable_sensor() (pyActLearn.CASAS.data.CASASData method), 10
EventHour (class in pyActLearn.CASAS.stat_features), 19
EventSeconds (class in pyActLearn.CASAS.stat_features), 19
export_fuel() (pyActLearn.CASAS.data.CASASData method), 11
export_hdf5() (pyActLearn.CASAS.data.CASASData method), 11

F

FeatureRoutineTemplate (class in pyActLearn.CASAS.stat_features), 21
FeatureTemplate (class in pyActLearn.CASAS.stat_features), 20
fetch_data() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16
flush() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16

G

get_activities_by_indices() (pyActLearn.CASAS.data.CASASData method), 11
get_activity_by_index() (pyActLearn.CASAS.data.CASASData method), 11
get_activity_color() (pyActLearn.CASAS.data.CASASData method), 11
get_activity_index() (pyActLearn.CASAS.data.CASASData method), 11
get_bg_target() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16
get_bg_target_id() (pyActLearn.CASAS.h5py.CASASHDF5

method), 16
get_confusion_matrix() (in module pyActLearn.performance), 22
get_days_info() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16
get_enabled_activities() (pyActLearn.CASAS.data.CASASData method), 11
get_enabled_sensors() (pyActLearn.CASAS.data.CASASData method), 12
get_feature_by_index() (pyActLearn.CASAS.data.CASASData method), 12
get_feature_description_by_index() (pyActLearn.CASAS.h5py.CASASHDF5 method), 16
get_feature_string_by_index() (pyActLearn.CASAS.data.CASASData method), 12
get_feature_value() (pyActLearn.CASAS.stat_features.DominantSensor method), 20
get_feature_value() (pyActLearn.CASAS.stat_features.EventHour method), 19
get_feature_value() (pyActLearn.CASAS.stat_features.EventSeconds method), 19
get_feature_value() (pyActLearn.CASAS.stat_features.FeatureTemplate method), 21
get_feature_value() (pyActLearn.CASAS.stat_features.LastSensor method), 18
get_feature_value() (pyActLearn.CASAS.stat_features.SensorCount method), 20
get_feature_value() (pyActLearn.CASAS.stat_features.SensorElapseTime method), 20
get_feature_value() (pyActLearn.CASAS.stat_features.WindowDuration method), 18
get_performance_array() (in module pyActLearn.performance), 22
get_sensor_by_index() (pyActLearn.CASAS.data.CASASData method), 12
get_sensor_by_index() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
get_sensor_index() (pyActLearn.CASAS.data.CASASData method), 12
get_target_color_by_index() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
get_target_description_by_index() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
get_target_descriptions() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
get_weeks_info() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17

I

is_bg_target() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
is_value_valid (pyActLearn.CASAS.stat_features.FeatureTemplate attribute), 21

L

LastSensor (class in pyActLearn.CASAS.stat_features), 18

N

num_between_splits() (pyActLearn.CASAS.h5py.CASASHDF5 method), 17
num_features() (pyActLearn.CASAS.h5py.CASASHDF5 method), 18
num_sensors() (pyActLearn.CASAS.h5py.CASASHDF5 method), 18
num_targets() (pyActLearn.CASAS.h5py.CASASHDF5 method), 18

P

per_class_event_scoring() (in module pyActLearn.performance.event), 24
per_class_segment_scoring() (in module pyActLearn.performance.event), 25
populate_feature() (pyActLearn.CASAS.data.CASASData method), 12
pyActLearn.CASAS.data (module), 9
pyActLearn.CASAS.h5py (module), 15

S

score_segment() (in module pyActLearn.performance.event), 24
SensorCount (class in pyActLearn.CASAS.stat_features), 19
SensorElapseTime (class in pyActLearn.CASAS.stat_features), 20
set_background_target() (pyActLearn.CASAS.h5py.CASASHDF5 method), 18

summary() (pyActLearn.CASAS.data.CASASData method), 12

U

update() (pyActLearn.CASAS.stat_features.FeatureRoutineTemplate method), 21

W

WindowDuration (class in pyActLearn.CASAS.stat_features), 18
write_to_xlsx() (pyActLearn.CASAS.data.CASASData method), 12